**VŠB – Technická univerzita Ostrava**
**Fakulta elektrotechniky a informatiky**
**Katedra informatiky**

# Simulační platforma pro analýzu finančních trhů

# Simulation Platform for Financial Markets Analysis

**2016** **Robert Saniga**

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

# Zadání diplomové práce

Student: **Bc. Robert Saniga**

Studijní program: N2647 Informační a komunikační technologie

Studijní obor: 2612T025 Informatika a výpočetní technika

Téma: Simulační platforma pro analýzu finančních trhů
Simulation Platform for Financial Markets Analysis

Jazyk vypracování: čeština

Zásady pro vypracování:

Cílem této diplomové práce je návrh a implementace simulační platformy pro analýzu obchodování akciových trhů. Simulační platforma bude složena z obchodních agentů, kteří v reálném čase budou nakupovat či prodávat akcii v závislosti na jejich implementovaném chování (market maker a market taker). Platforma bude obsahovat následující vlastnosti:
1. Zpracování nákupních a prodejních příkazů od obchodních agentů v reálném čase.
2. Monitorování událostí simulační platformy.
3. Záznam historie obchodování agentů pro možnost analýzy časové řady akciového trhu.

Seznam doporučené odborné literatury:

[1] Tirea, M.; Tandau, I.; Negru, V. "Multi-agent Stock Trading Algorithm Model", Symbolic and Numeric Algorithms for Scientific Computing (SYNASC), 2011 13th International Symposium on, On page(s): 365 - 372
[2] Luo, Yuan, Darryl N. Davis, and Kecheng Liu. "A multi-agent system framework for decision support in Stock Trading." IEEE Network Magazine Special Issue on Enterprise Networking and Services (2002).
[3] Hintjens, Pieter. ZeroMQ: Messaging for Many Applications. " O'Reilly Media, Inc.", 2013.
[4] Akgul, Faruk. ZeroMQ. Packt Publishing Ltd, 2013.

Formální náležitosti a rozsah diplomové práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí diplomové práce: **Ing. Lukáš Zaorálek**

Datum zadání: 01.09.2014
Datum odevzdání: 29.04.2016

VYSOKÁ ŠKOLA BÁŇSKÁ-TECHNICKÁ UNIVERZITA * FAKULTA elektrotechniky a informatiky * OSTRAVA *

doc. Dr. Ing. Eduard Sojka
*vedoucí katedry*

prof. RNDr. Václav Snášel, CSc.
*děkan fakulty*

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 29.04.2016

.....

Na tomto místě bych rád poděkoval svému vedoucímu diplomové práce panu Ing. Lukáši Zaorálkovi za jeho investovaný čas a odbornou pomoc.

## Abstrakt

Cílem této diplomové práce je navrhnout a implementovat platformu pro simulaci a analýzu obchodování na akciových trzích. Platforma je pojata jako multiagentní systém, ve kterém hlavní části akciového trhu tvoří autonomní agenti komunikující mezi sebou pomocí obchodních příkazů a událostí. Agent typu obchodník vytváří na základě zvolených obchodních strategií jednotlivé obchodní příkazy a následně je posílá prostřednictvím obchodní brány do trhu. Trh po vyhodnocení příkazu vytváří odpovídající obchodní události. Z obchodních událostí jsou počítány základní statistiky, které obchodník může dále využít pro svůj Money-management. Všechny obchodní příkazy a události platformy jsou monitorovány a ukládány pro pozdější analýzu. Cílem práce je rovněž měření výkonnosti takového řešení a jeho vyhodnocení.

## Klíčová slova

Simulační platforma, akciové trhy, akcie, matematický model akcie, Order Book, multiagentní systémy, Order Matching Engine, ZMQ, Protocol Buffers, Java

## Abstract

The aim of this thesis is to design and implement a platform for simulation and analysis of trading on the stock markets. The platform is designed as a multi-agent system in which the major part of the stock market consists of autonomous agents communicating between themselves by trading orders and events. Agent type Trader forms individual trading orders based on the selected trading strategies and then sends them through a market gateway to the market. Market after evaluating the order creates corresponding market events. Based on market events are counted basic statistics that the trader may use further for Money-management. All trading orders and platform events are monitored and stored for later analysis. The aim is to also measure the performance of such a solution and its evaluation.

## Key Words

Simulation platform, Stock Markets, Share, Mathematical Model of Stock, Order Book, Multi-agent systems, Order Matching Engine, ZMQ, Protocol Buffers, Java

# Obsah

# 1 Seznam ilustrací

# 2 Úvod

Rozmach informačních technologií v posledních desetiletích znamená pro mnohá odvětví významnou změnu ve vnímání a využívání moderních přístupů založených na informačních systémech. Stejně tak je tomu v oblasti finančních trhů. Tam kde dříve stávali obchodníci dohadující se nad cenami obchodů, jsou nyní výkonné stroje obsluhující desetitisíce obchodních příkazů za sekundu. To zavdalo příležitost vzniknout také poměrně mladému oboru zabývajícímu se automatizovanými obchodními systémy (*Automated Trading System* - ATS). Ty na základě naprogramovaného chování přebírají úlohu obchodování za lidi. Speciálním případem takových algoritmů je *High Frequency Trading* (HFT), obchodování s vysokou frekvencí zadávání příkazů. Ty prodávají nebo nakupují obrovské objemy akcií (popř. jiného obchodního instrumentu) ve zlomcích sekundy s cílem malých jednotlivých zisků, vykonáváním tohoto procesu neustále dokola však vede k vysokým celkovým ziskům.

Pro vývoj HFT strategií je nutné takové strategie testovat, ověřovat jejich funkčnost a profitabilitu, a právě to bylo příčinou vzniku této diplomová práce. Práce si klade za cíl vytvořit takové prostředí, ve kterém můžeme proti sobě postavit skupinu obchodních strategií a nechat je mezi sebou „soutěžit“. Pro vyhodnocení ziskovosti jednotlivých strategií jsou vedeny statistiky počítané z událostí vzniklých samotným obchodováním. Strategie pak mohou statistiky zvažovat a inteligentně jim přizpůsobit své chování. Proto lze na takový systém pohlížet rovněž jako na multiagentní systém sestávající se z inteligentních agentů obchodníků komunikujících mezi sebou navzájem skrze obchodní příkazy.

Práce začíná částí teoretickou, ve které čtenáře nejprve seznamuje se základními termíny z oblasti finančních trhů a jak finanční trhy fungují (kapitola *3 Finanční trhy a obchodování*), dále pak je to úvod do multiagentních systémů v kapitole *4 Multiagentní systémy* a v kapitole *5 Order Matching Engine* je popsáno jakým způsobem se vyhodnocují obchodní příkazy přicházející na trh. Kapitola *6 Komunikační (transportní) vrstva* popisuje základní kámen navrhovaného multiagentního systému a jaké prostředky k němu využijeme (knihovna ZMQ). V kapitole *7 Serializace dat* jsou popsány a vyhodnoceny možnosti serializace dat pro transportní vrstvu. Kapitola *8 Monitorování* vyhodnocuje neméně důležitou vlastnost navrhované platformy, monitoring. Část praktická zahrnuje kapitolu *9 Návrh a implementace* a kapitolu *10 Testování výkonnosti platformy*. Práce je pak vyhodnocena v kapitole *11 Závěr*.

# 3 Finanční trhy a obchodování

Trh je místo, kde se setkává nabídka a poptávka po různých produktech, které zde nabízejí subjekty trhu. Finanční trh představuje souhrn nástrojů, postupů a institucí, jejichž prostřednictvím dochází k přelévání volných finančních prostředků od těch, kteří je vlastní, k těm, kteří jimi chtějí disponovat.
Finanční trhy již nejsou doménou jen vybraných obchodníků, kteří obchodují na tzv. *parketu* konkrétních burz, naopak většina obchodů se dnes realizuje elektronicky. Na obchodování lze pohlížet jako na podnikání s absolutním řízením, omezenými riziky a přitom s možností vysokých zisků. Ale i ztrát. Mezi nejčastější obchodní instrumenty patří [1]:

- Akcie
- Futures a komodity
- Opce
- Forex

Na obchodování je zajímavé, že vydělávat můžeme na růstu ale i na poklesu cen. To závisí na způsobu obchodování a tzv. *spekulování*. Spekulanti nekupují dané aktivum (investiční nástroj) na základě potřeby, ale za účelem zisku při pohybu ceny. Jsou běžnou součástí trhu a přináší sebou i potřebnou likviditu. Při spekulaci obchodník otevírá a uzavírá takzvané pozice. Ty se dělí na:

**Dlouhá (long) pozice**
Je spekulace, kdy očekáváme růst ceny nějaké komodity či cenného papíru. V tomto případě si investor předmět spekulace nejprve zakoupí (jde takzvaně long) a později se jej snaží pokud možno za vyšší cenu prodat. Pokud cena mezi nákupem a prodejem opravdu vzroste, investor zaznamená zisk. V opačném případě to však pro něj znamená ztrátu.
Obchodníkům spekulujícím na růst ceny se také říká „býci“ (býk útočí rohy zespoda nahoru, tedy stejně jako se vyvíjí trend ceny).

**Krátká (short) pozice**
Je spekulace na pokles ceny určité komodity či cenného papíru. Investor si nejprve za úplatu předmět spekulace půjčí a prodá třetí straně. Později jej investor nakoupí a vrátí zpět tomu, od koho si jej vypůjčil. Pokud cena mezi prodejem a zpětným nákupem klesla, zaznamená investor zisk. Nárůst ceny znamená ztrátu.
Obchodníkům, kteří spekulují na pokles ceny, se říká „medvědi“ (medvěd útočí tlapou shora dolů).

# 3.1 Základní typy příkazů na finančních trzích

Příkazy na finančních trzích se zpravidla vykonávají přes prostředníky, tzv. brokery. Ten zajišťuje styk mezi kupujícím a prodávajícím. Základními příkazy jsou tržní příkaz, neboli *Market*, a limitní příkaz, zkráceně *Limit*.

**Tržní příkaz (Market)**
Pokyn k nákupu či prodeji požadovaného množství lotů, kontraktů nebo množství cenných papírů za aktuálně nejlepší nabídku/poptávku v době doručení pokynu na trh. Jedná se o nejčastější příkaz, kdy není omezena cena, za niž může být příkaz naplněn. Příkaz je používán v případě, že investor chce nakoupit/prodat bezpodmínečně za aktuálně nejlepší cenu dostupnou na trhu.

**Limitní příkaz (Limit)**
Příkaz umístěný brokerem k nákupu, či prodeji určitého množství lotů, kontraktů či cenných papírů za specifikovanou cenu nebo lepší. Z tohoto důvodu se může stát, že příkaz nebude naplněn. Protože je plnění příkazu také složitější, vybírá si za něj brokerská společnost vyšší poplatky.

# 3.2 Order Book

neboli také Limit Order Book (LOB) je prostředek pro zaznamenávání nákupních a prodejních příkazů typu limit na elektronických trzích. Definice LOB (stejně jako jiných ekonomických pojmů) není jednotná a liší se s určitými detaily autor od autora [2][3]. LOB uchovává dosud nespárované limitní příkazy pro poptávku (tzv. bid) a nabídku (tzv. ask). Příkazy bývají uspořádány primárně podle cenové, sekundárně podle časové priority. V případě poptávky od ceny nejvyšší, v případě nabídky od ceny nejnižší. Příchozí příkazy typu Market jsou vždy párovány proti nejlepším příkazům v LOB. Tyto příkazy uspořádaně definují nejlepší ceny na trhu. LOB lze matematicky definovat jako dvojici vektorů:

$$LOB = (a(t); b(t)) = (a_1(t), ..., a_n(t); b_1(t), ..., b_m(t)) \tag{3.1}$$

Kde $a = (a_1, ..., a_n)$ určuje stranu nabídky LOB a $a_i$ značí počet akcií na cenové úrovni $i$. $b = (b_1, ..., b_m)$ obdobně určuje stranu poptávky LOB a $b_i$ značí počet akcií na cenové úrovni $i$.
Vzdálenost mezi nejlepší nabídkou a nejlepší poptávkou se nazývá *spread*. Pokud nejlepší (nejnižší) cenu nabídky označíme $P^A$ a nejlepší (nejvyšší) cenu poptávky označíme $P^B$, můžeme spread $S$ matematicky definovat jako:

$$S = P^A - P^B \tag{3.2}$$

Rovnice

$$P = \frac{P^A + P^B}{2} \tag{3.3}$$

definuje takzvanou střední hodnotu $P$ (*mid-price*), která bývá obvykle uváděna společně s nejlepšími cenami poptávky a nabídky.
Na základě LOB a jeho vývoje můžeme modelovat vztah mezi nabídkou a poptávkou, vývoj ceny, určit cenu finančního instrumentu (označovaný jako kurz), volatilitu a jiné.

# 3.3 Akciové trhy

Akciový trh je druhem kapitálového trhu. Společnosti zde nabízejí akcie, aby získaly prostředky k financování svého chodu, rozvoje, výzkumu. Vlastnictvím akcie se majitel stává společníkem (akcionářem), s čímž jsou spojena práva akcionáře:

- může se podílet na řízení společnosti – právo účasti a hlasování na valné hromadě akcionářů,
- podílí se na zisku společnosti – právo na dividendy,
- náleží mu podíl na likvidačním zůstatku.

Akciový trh obecně je místo, kde se setkávají nákupci a prodejci akcií. Toto zahrnuje jednak akcie prodávané veřejně, tedy na tzv. burzách cenných papírů, ale i akcie prodávané pouze privátně. Pojmem akciové trhy se tedy v této diplomové práci rozumí právě místo, kde se akcie prodávají veřejně na burzách.
Burza cenných papírů je místo nebo také organizace, skrze kterou mohou tzv. prodejci akcií - což mohou být jednak samostatné osoby, ale i velké firmy – obchodovat akcie. Firmy (akciové společnosti) mohou také své akcie nabízet na burzách. Velké firmy obvykle nabízí své akcie na více burzách celého světa (např. Coca-Cola, Nestlé, …).
Mezi největší akciové trhy patří New York Stock Exchange na Wall Street v New Yorku, NASDAQ (nachází se také v New Yorku a je pouze elektronická, tedy všechny transakce zde probíhají pouze přes počítač a Internet), London Stock Exchange Group nacházející se v Londýně. Největší českou burzou cenných papírů je pražská BCPP (Burza cenných papírů Praha). [3]
Obchod na akciových trzích znamená převod peněz potřebných pro nákup akcií od nakupujícího k prodávajícímu na dohodnuté ceně za akcii. Akcie se pak stávají majetkem nakupujícího.
Akcie mají hodnotu nominální, tržní, dividendovou. Nominální hodnotou se rozumí podíl na majetku akciové společnosti a tedy součet nominálních hodnot všech akcií jedné společnosti je roven výši základního jmění této společnosti. Dividendovou hodnotou je podíl na zisku společnosti. Cena, se kterou se obchoduje na trhu, je pak hodnotou tržní. Cena akcie bývá také označována jako kurz akcie.
Kurz akcie je ovlivňován jak nabídkou a poptávkou akcie dané akciové společnosti na burze cenných papírů, tak například také pozitivní či negativní zprávou o dané akciové společnosti, informacemi o vývoji hospodářského odvětví, ekonomiky a dalšími (i nejen čistě ekonomickými) faktory.

## 3.3.1 Klasifikace akcií

Akcie lze klasifikovat podle mnoha faktorů. Tyto faktory udávají některé vlastnosti akcií jako je například volatilita, likvidita, rizikovost. Zde uvedu jen nejznámější typy akcií, o kterých čtenář mohl někdy zaslechnout.

**Blue chips**
Pojmenování pro skupinu dobře zavedených společností s vysokou tržní kapitalizací. Mají dobrou pověst, dlouhodobou profitabilitu a výplatu dividend. Jedná se o stabilní typ akcií.

**Penny akcie (penny stock)**
Levné spekulativní akcie prodávané za méně než 5 dolarů za akcii. Penny akcie nabízí společnosti s nízkou tržní kapitalizací a jsou obvykle obchodovány mimo hlavní burzy na tzv. *Over-The-Counter* (OTC) trzích. Penny akcie mívají vysokou volatilitu, a tak se zde otevírá možnost vysokých zhodnocení (až stovky procent).

**Klasifikace podle tržní kapitalizace**
Tržní kapitalizace společnosti je dána počtem akcií v oběhu a cenou jedné akcie [1]:

- Akcie s mega kapitalizací (mega-cap stock) – je-li kapitalizace vyšší než 200 miliard dolarů
- Akcie s velkou kapitalizací (large-cap stock) – kapitalizace v rozmezí 10-200 miliard dolarů
- Akcie se střední kapitalizací (mid-cap stock) – kapitalizace 2-10 miliard dolarů
- Akcie s malou kapitalizací (small-cap stock) – kapitalizace 300 milionů až 2 miliardy dolarů
- Akci s nano kapitalizací (nano-cap stock) – kapitalizace nižší než 50 milionů dolarů

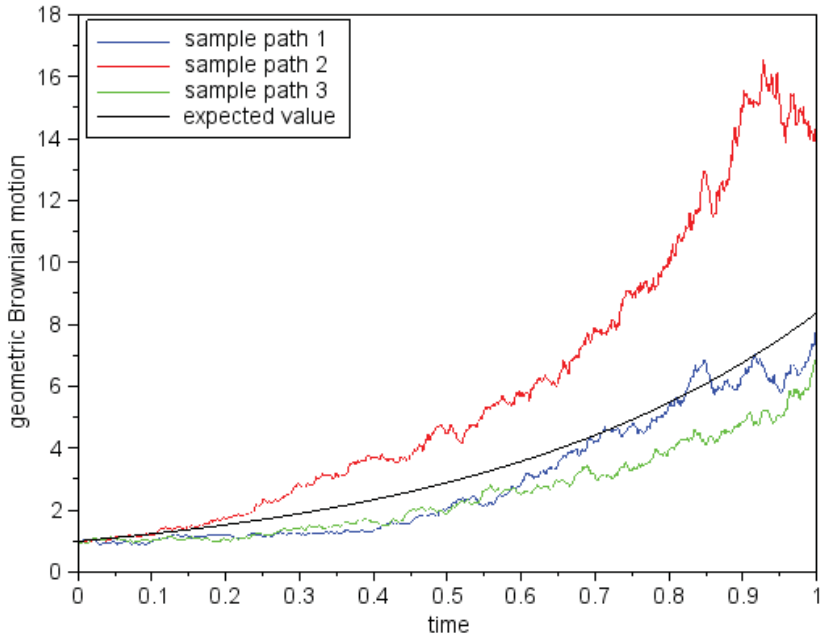
## 3.4 Matematický model akcií

Z pohledu matematiky můžeme na akcie pohlížet rovněž jako na Brownův pohyb, konkrétně jeho geometrickou (či exponenciální) verzi. Tu lze vyjádřit stochastickou diferenciální rovnicí [4]:

$$dS_t = \mu S_t dt + \sigma S_t dW_t \tag{3.4}$$

, kde
$S_t$ – cena akcie v čase $t$
$\mu$ – procentuální vyjádření trendu (*drift*), úroková míra
$\sigma$ – procentuální vyjádření volatility (rozptyl)
$W_t$ – Wienerův proces (Brownův pohyb)

Z pohledu trajektorie Brownova pohybu vyjadřuje levá složka rovnice $\mu S_t dt$ celkový trend trajektorie, zatímco pravá složka rovnice $\sigma S_t dW_t$ vyjadřuje šum trajektorie způsobený neočekávanými událostmi během pohybu.
Na Obrázek 3.1 dále můžeme sledovat 3 různé průběhy geometrického Brownova pohybu se stejným nastavením konstant $\mu$, $\sigma$ a počáteční hodnoty $S_0$. I když tvar jednotlivých grafů se liší, jejich trend je shodný – vyjádřený průběhem *expected value*. Různý průběh grafů je dán právě stochastickou složkou $W_t$ – Brownovým pohybem.
I když je geometrický Brownův pohyb nejpoužívanějším přístupem k modelování pohybu cen akcií, realitě zcela neodpovídá. Realitě neodpovídá konstantní hodnota volatility $\sigma$, která se ve skutečnosti mění s časem, a spojitý pohyb ceny akcie. Cena akcie se může měnit skokem. Tento přístup se používá pro svou relativní jednoduchost. [4]




*Obrázek 3.1: Ukázkové průběhy geometrického Brownova pohybu*[1]

Brownův pohyb má z matematického hlediska ještě jednu zajímavou vlastnost. Její budoucí stav je závislý pouze na stavu aktuálním, není nijak závislý na stavech minulých. Je tedy bezpaměťový. Tomuto jevu se říká Markovova vlastnost (*Markov property*). Brownův pohyb je tedy také Markovův proces. Pro dokazování této vlastnosti exaktně (matematicky) je nutné nejprve vyřešit diferenciální rovnici popisující geometrický Brownův pohyb. Po úpravě vyjde řešení [5]:

$$S(t) = S_0 e^{X(t)} \tag{3.5}$$

, kde $X(t) = \mu t + \sigma W(t)$

Nyní můžeme přidáním $h$ časových jednotek prokázat výskyt Markovovi vlastnosti:

$$\begin{aligned} S(t+h) &= S_0 e^{X(t+h)} \\ S(t+h) &= S_0 e^{X(t)+X(t+h)-X(t)} \\ S(t+h) &= S_0 e^{X(t)} e^{X(t+h)-X(t)} \\ S(t+h) &= S(t) e^{X(t+h)-X(t)} \end{aligned} \tag{3.6}$$

[1] Obrázek převzat z http://www.iam.fmph.uniba.sk/institute/stehlikova/fd14en/ex/ex02.html

Při daném *S(t)* závisí budoucí *S(t + h)* pouze na budoucím inkrementu Brownova pohybu, konkrétně na *X(t + h) - X(t)*. Protože Brownův pohyb má pouze nezávislé inkrementy (jedná se o stochastický jev), je budoucí stav nezávislý na stavu minulém => Markovova vlastnost.

### 3.4.1 Párové obchodování

Na základě výše uvedeného matematického aparátu je možné exaktně pracovat s pojmem akcie (cena akcie) a jejími dalšími vlastnostmi a využívat je tak k implementaci obchodních (investičních) strategií. Jednou z takových strategií, těšící se velké oblibě u například hedgeových fondů, je takzvané Párové obchodování (*Pairs trading*).

Párové obchodování je založeno na identifikaci dvojic (párů) akcií, které se obvykle obchodují v určitém vzájemném a hlavně předvídatelném vztahu. Když se u takového páru objeví nějaká odchylka (cena jedné akcie prudce stoupne/klesne oproti druhé), strategie Párového obchodování uzavře kontrakty s cílem zisku při návratu těchto akcií do běžného poměru.

Příkladem mohou být vysoce korelující akcie stejného (průmyslového) odvětví, jako např. mezi akciemi společností Coca-Cola a Pepsi. Akcie v takovém páru bývají obchodovány relativně jedna ke druhé. Zvýší-li se cena akcií společnosti Coca-Cola oproti ceně akcií společnosti Pepsi, strategie Párového obchodování by předpokládala návrat do „normálu“ a velela by k nákupu akcií společnosti Pepsi a naopak prodeje akcií Coca-Cola. Při návratu do normálního poměru - a při správně zvoleném poměru mezi počtem prodaných a nakoupených akcií (hodnota prodaných akcií by měla být shodná s hodnotou nakoupených akcií) - mezi těmito akciemi by strategie Párového obchodování zaznamenala profit. [4]

Párové obchodování se opírá o složité matematické nástroje a pojmy, které překračují rámec této práce a které by vystačily na samostatnou diplomovou práci. Proto zde uvedu jen jeden z nich, Ornstein-Uhlenbeck process.

**Ornstein-Uhlenbeck process**

je stochastický proces, který je stacionární (na rozdíl od Wienerova procesu), má Gaussův průběh (normální distribuci) a je Markovovský (budoucí stav je dán pouze stavem současným, je nezávislý na stavech předchozích). [7][8]

Lze jej vyjádřit lineární diferenciální rovnicí:

$$dX_t = -\rho(X_t - \mu)dt + \sigma dW_t \tag{3.7}$$

, kde $\rho$, $\mu$, $\sigma$ jsou konstanty vyjadřující průměr, volatilitu a rychlost návratu k průměrné hodnotě a $W_t$ je starý známý Wienerův proces. [9][10][11]

V párovém obchodování slouží k modelování spreadu. [4]

Na základě Ornstein-Uhlenbeck procesu je také například založeno modelování vývoje úrokové míry. [12]

# 4 Multiagentní systémy

V následující kapitole bude popsána architektura multiagentního systému, na jehož základech je postaven navrhovaný systém této diplomové práce.
Konkrétně bude popsána klíčová koncepce multiagentního systému, následována rozdělením základních architektur a sekcí týkající se programovacích jazyků a nástrojů. Zmíněna bude také organizace FIPA, která stála na počátku standardizace tohoto odvětví.

## 4.1 Agent

je speciální softwarová komponenta mající určitou úroveň autonomie a poskytující rozhraní, přes které je možno s tímto agentem komunikovat. Tak jako v reálném světě se agent snaží splnit přání svého klienta. Výrazu „agent“ ve smyslu softwarového agenta se běžně využívá v oblasti umělé inteligence, databází, operačních a počítačových systémů.
I když agentní systém může obsahovat pouze jediného agenta, který pracuje sám ve svém prostředí a případně komunikuje s uživateli, obvykle se skládá z více navzájem komunikujících agentů. V tomto případě hovoříme o multiagentních systémech (MAS). Ty mohou tvořit komplexní systémy, kde jednotlivý agenti mají společné nebo navzájem konfliktní cíle. Agenti zde spolu interagují přímo (komunikace a vyjednávání), či nepřímo (prostřednictvím prostředí, ve kterém se nachází). Agenti mohou spolupracovat na společném cíli, ale také si mohou navzájem konkurovat pro dosažení vlastních zájmů. Agenty lze charakterizovat těmito vlastnostmi:

- **autonomní** – vykonává svou činnost bez zásahu člověka nebo jiných a má kontrolu nad svými akcemi a stavem,
- **sociální** – kooperuje s lidmi a/nebo jinými agenty pro vykonání svých úkolů,
- **reaktivní** – reaguje na změny v prostředí, ve kterém se nachází,
- **proaktivní** – kromě akcí spojených se změnou prostředí, je také schopen převzít iniciativu pro dosažení svých cílů.

Dále mohou agenti být také:

- **mobilní** – mají schopnost přesouvat se mezi různými uzly v počítačové síti,
- **pravdomluvný** – poskytuje jistotu, že nebude úmyslně podávat nepravdivé informace,
- **benevolentní** – vždy se snaží vykonat, o co je požádán,
- **racionální** – vždy vystupuje s cílem dosáhnout svých cílů a nikdy s cílem jim zabránit,
- **učenlivý** – adaptující se svému prostředí a přáním svých uživatelů.

Agentově-orientované programování je softwarové paradigma, které přináší koncepce z teorie umělé inteligence do oblasti distribuovaných systémů. Agentové pojetí coby základního kamene systému mění přístup ke konceptuálnímu návrhu a implementaci software.
Multiagentní systémy a agentová technologie obecně se v posledních letech dostává do popředí zájmu mnoha odvětví. Jsou to aplikace jak na straně malých systému pro osobní použití (mobilní agent, který kompenzuje dopad slabé konektivity kešováním dat) až k systémům složitým pro průmyslovou aplikaci

jako je logistika, systémová diagnostika, řízení výroby, robotika, ale také například management v komunikačních a počítačových sítích.
Agentové systémy také přináší nový přístup k vyřešení integrace starých softwarových (tzv. *legacy*) systémů.

# 4.2 Architektura

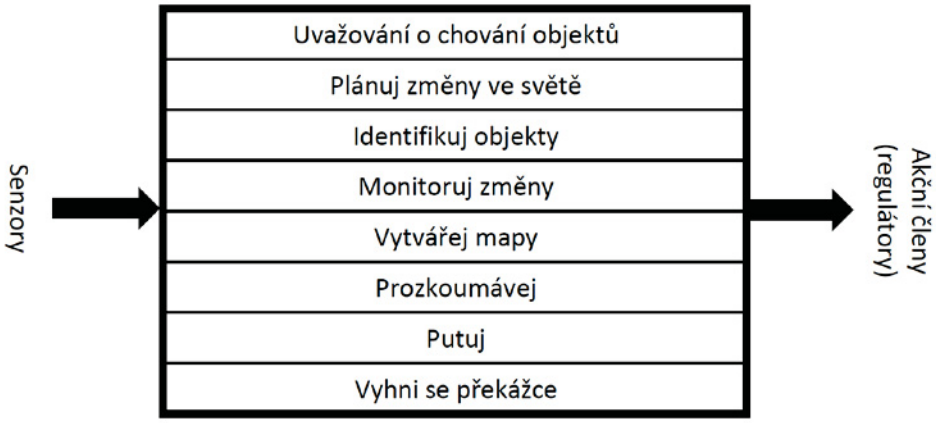
Architektury multiagentních systémů se vyvíjely od těch jednoduchých, založených na čistě reaktivním přístupu (formou stimul-reakce), až po opačný pól, kdy agenti zvažují důvody svých akcí, jako je tomu v případě modelu BDI. Mezi těmito dvěma extrémy leží hybridní přístup, který se snaží využít výhod obou architektur. Architektura multiagentních systémů může být rozdělena do 4 hlavních skupin – logická, reaktivní, BDI a vícevrstvá architektura.

## 4.2.1 Logická (*logic-based*)

Architektura založená na reprezentaci světa pomocí logických symbolů a mechanismů. Výhodou je, že tato reprezentace je velice blízká lidskému chápání. Na druhou stranu taková reprezentace reálného světa může být velice složitá, a tedy zpracování aplikačního kódu může být časově náročné. Výsledky mohou být k dispozici až příliš pozdě na to, aby byly stále použitelné.

## 4.2.2 Reaktivní

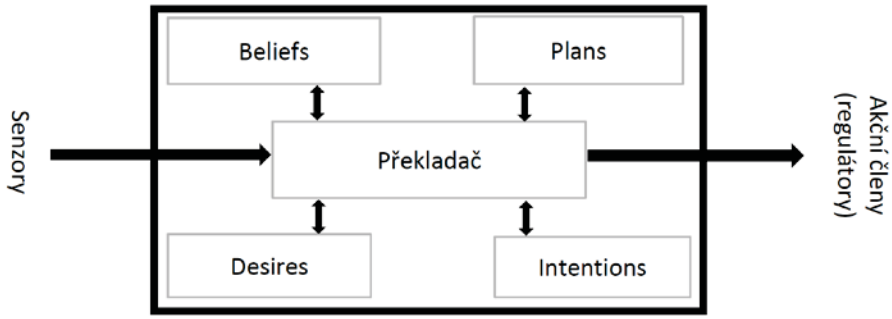
Reaktivní architektura implementuje rozhodovací mechanismus jako přímé mapování situací (stimulů) na akce. Je založena na jednoduchém mechanismu stimul-odpověď, který je vyvoláván daty ze senzorů. Na rozdíl od logické architektury nemá symbolickou reprezentaci okolního světa, a tedy není zatěžována komplexním uvažováním.
Jednou z nejvíce známých reaktivních architektur je subsumpční architektura představená Rodneyem Brooksem. Subsumpční architektura definuje vrstvy sestávající se z konečných automatů. Jednotlivé vrstvy jsou přímo připojeny k senzorům, ze kterých získávají reálná data a na které mohou přímo reagovat naprogramovaným chováním. Vrstvy tak tvoří jakousi hierarchii chování, kde nižší vrstvy zásobníku mají více kontroly než vrstvy vyšší. Příkladem je navigace robota na Obrázek 4.1.
Na jednoduchém návrhu pro navigaci robota vidíme, že například chování „Vyhni se překážce“ má vyšší prioritu než chování „Prozkoumávej“ (určitě je pro nás důležitější, aby nedošlo k fyzickému poškození robota než jeho schopnost průzkumu ve chvíli, kdy narazí na překážku).
Výhodou reaktivních agentů je jejich lepší přizpůsobení dynamickým prostředím, kde reagují zpravidla rychleji než v případě architektury logické. Další výhodou je jejich jednodušší návrh. Tato výhoda se v některých případech však stává jejich nevýhodou. Jednodušší návrh je předurčuje k aplikaci pouze pro určité typy úkolů a téměř vylučuje agenty s vysokým počtem chování. V některých případech data získávaná ze senzorů mohou být nedostatečná pro vhodné rozhodování.




*Obrázek 4.1: Příklad subsumpční architektury pro navigaci robota*

### 4.2.3 BDI

Název BDI je zkratkou 3 slov: přesvědčení (*Belief*), přání (*Desire*) a záměr (*Intention*). Tento přístup, který má mimo jiné původ ve filozofii, je postaven na modální logice.
Jedna z nejznámějších BDI architektur je tzv. *Procedural Reasoning System* (PRS). Ten se skládá ze 4 základních datových struktur (*beliefs*, *desires*, *intentions* a *plans*) a překladače (viz Obrázek 4.2).




*Obrázek 4.2: Procedural Reasoning System (PRS)*

Datová struktura *beliefs* reprezentuje informace agenta o svém prostředí, která však mohou být nekompletní či nepřesná. *Desires* reprezentuje úkoly (cíle) a *intentions* jsou již agentem schválené úkoly. *Plans* specifikují některé akce, které agent bude vykonávat pro dosažení *intentions*. Datové struktury jsou řízeny překladačem. Ten je zodpovědný za aktualizaci *beliefs* v závislosti na pozorování prostředí, generování nových úkolů (*desires*) na základě nových *beliefs* a povyšování některých *desires* na *intentions*.

### 4.2.4 Vícevrstvá (*layered*)

neboli také hybridní architektura, která umožňuje obojí. Jak reaktivní, tak deliberativní přístup tím, že se systém skládá z více vrstev. Vrstvy mohou být skládány horizontálně (jednodušší přístup, který však může vést k příliš vysokému počtu interakcí mezi jednotlivými vrstvami), nebo vertikálně (eliminuje zmíněné nevýhody horizontálního přístupu, ale datový tok je sekvenční a systém tak není tzv. *fault tolerant*[2]).

## 4.3 Programovací jazyky a nástroje

Správný výběr programovacího jazyka a softwarových knihoven (popř. i vývojového nástroje) patří neodmyslitelně k dalšímu důležitému aspektu při realizaci multiagentních systémů.
I když lze multiagentní systémy implementovat pomocí libovolného programovacího jazyka, objektově orientované jazyky jsou upřednostňovány. Koncept pojmu agent není tolik vzdálen od pojmu objekt. Také multiagentní systémy často používají zapouzdření logiky, dědičnost, předávání zpráv.
Jedním z nejznámějších nástrojů pro vývoj multiagentních systémů je knihovna JADE, která je založena na platformě a programovacím jazyce Java.
Existuje však také celá nová třída programovacích jazyků, takzvané agentově-orientované jazyky, které jsou výhradně zaměřeny na vývoj agentových systémů. Příkladem jsou jazyky jako FLUX, 3APL, JACK Agent Language.

## 4.4 Foundation for Intelligent, Physical Agents

zkráceně FIPA, byla organizace založená pro vytváření specifikací a podporu technologií zabývajících se softwarovými agenty.
Jednou ze specifikací je i FIPA ACL (*Agent Communication Language*), jazyk pro komunikaci a výměnu informací mezi jednotlivými agenty. Jazyk definuje určitý protokol interakce, umožňuje však použít libovolný jazyk pro informační obsah.
Klíčovými koncepcemi FIPA specifikací jsou agentová komunikace, agentový managament a agentová architektura.

### 4.4.1 Agentová komunikace

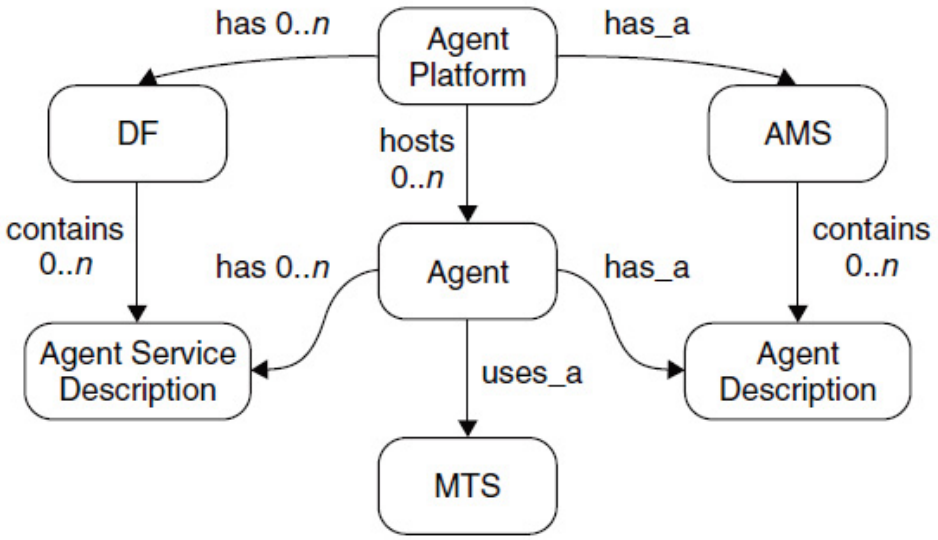
Z pohledu komunikace se multiagentní systém dělí na komponenty a konektory. Komponenty jsou konzumenty, producenty a prostředníky při předávání zpráv. Konektory slouží k výměně těchto zpráv, tvoří rozhraní.
FIPA pro agentovou komunikaci vytvořila tzv. *communication stack*[3], který se skládá z několika podvrstev. Tyto jsou začleněny do aplikační vrstvy klasického OSI modelu pro TCP/IP.

---

[2] pokud jedna vrstva selže, celý systém selže
[3] Souhrn komunikačních pravidel a protokolů, na kterých je technologie postavena

### 4.4.2 Agentový management

Druhý základní aspekt agentových systémů podle FIPA. Referenční model pro agentový management se skládá z několika komponent uvedených na Obrázek 4.3.




*Obrázek 4.3: FIPA referenční model pro agentový management* [13]

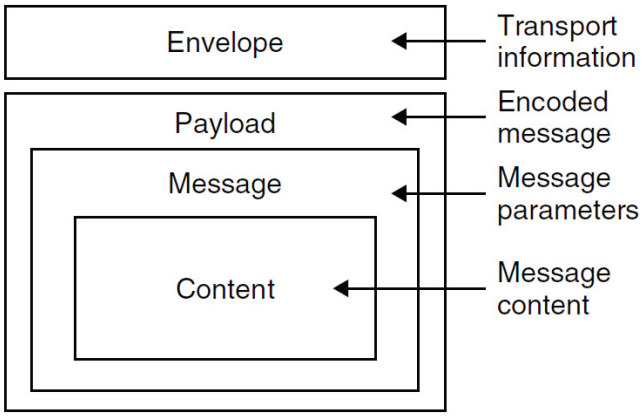
**Agent Platform (AP)** – infrastruktura, do níž jsou agenti nasazeni. Skládá se z počítačů, operačních systémů, dalších komponent podle FIPA specifikace pro agentový management, samotných agentů a popřípadě dalšího podpůrného software.

**Agent** – výpočetní proces poskytující nějakou službu. Agent musí mít alespoň jednoho vlastníka a musí být identifikovatelný takzvaným AID (FIPA *Agent Identifier*). Agent může být zaregistrován na více transportních adresách, přes které může být dále kontaktován.

**Directory Facilitator (DF)** – volitelná komponenta, která poskytuje informace o jednotlivých agentech (tzv. *yellow pages*). Měl by podporovat registraci, deregistraci a vyhledávání agentů a změnu o nich poskytovaných informací.

**Agent Management System (AMS)** – povinná komponenta zodpovídající za management operací, jako jsou např. vytváření, odstraňování, migrace agentů. Každý agent se musí u AMS registrovat, aby obdržel AID. AMS udržuje seznam agentů rovněž s jejich stavem (např. aktivní, ukončený, čekající). Poskytuje podobné služby jako DF (registraci, deregistraci, vyhledávání agentů). Má právo ukončovat jednotlivé operace agentů. V celém AP se vyskytuje pouze jediný AMS, pokud se AP rozprostírá přes více počítačů, je AMS autoritou napříč celou touto počítačovou sítí.

**Message Transport Service (MTS)** – služba poskytovaná AP k přenosu FIPA-ACL zpráv. Podporuje jak přenos zpráv mezi agenty v rámci jednoho AP, tak přenos mezi agenty různých AP. Obecnou strukturu takové zprávy podle FIPA popisuje následující Obrázek 4.4.




*Obrázek 4.4: Struktura zprávy podle FIPA* [13]

# 5 Order Matching Engine

V kapitole 3.2 jsem zmínil základní prvek elektronických finančních trhů – Order book. Tato kapitola se věnuje podrobnějšímu popisu fungování Order booku, zejména algoritmy pro zpracování obchodních (Limit a Market) příkazů. Na základě těchto algoritmů pak můžeme vytvořit komponentu Order Matching Engine, která v sobě zahrnuje Order book a řeší vypořádání jednotlivých příkazů.

## 5.1 Základní principy

Nad výše zmíněným Order bookem pracuje algoritmus, který srovnává a popřípadě páruje nově příchozí příkazy s těmi zbývajícími, ne zcela vyřízenými příkazy v Order booku. Páruje obchodní příkazy na prodej s těmi na nákup a naopak. Takovýmto algoritmům se říká „Matching Algorithms“. Algoritmů je celá řada a každý trh používá svůj specifický (například algoritmus pro opce je zpravidla odlišný od toho pro akcie). Cíl všech těchto algoritmů by však měl být stejný – poskytnout účastníkovy trhu nejlepší možné vypořádání za co „nejférovější cenu“.
Order matching má obecně 3 kroky:

1) Určit současnou cenu protější k příchozímu příkazu
2) Určit kvantitu protější k příchozímu příkazu
3) Alokovat tuto kvantitu pro obchod

Mezi základní algoritmy patří *FIFO* (first-in, first-out) algoritmus – alokuje kvantitu čistě na základě časové priority – a *Pro-Rata* algoritmus. Dále si znázorníme jejich chování.

## 5.2 FIFO

FIFO algoritmus poskytuje prioritně úplnou alokaci nejstaršímu příkazu na nejlepší cenové hladině bid/ask. Teprve pokud je vždy nejstarší příkaz plně vypořádán, přistupuje se k alokaci dalšího příkazu v pořadí podle času. Mechanismus si ukážeme na následujícím příkladu (Obrázek 5.1).
Mějme Order Book, ve kterém se nachází 3 nevyřízené příkazy na nákup různého počtu lotů na stejné cenové hladině. Jako další přichází příkaz k prodeji velkého počtu lotů na stejné cenové hladině.

| Čas. razítko | Příkaz # | BUY/SELL | Počet lotů | Cena |
|---|---|---|---|---|
| t | S1 | SELL | 200 | 50,90 |

**Order Book**

| Čas. razítko | Příkaz # | BUY/SELL | Počet lotů | Cena |
|---|---|---|---|---|
| t-3 | B1 | BUY | 100 | 50,90 |
| t-2 | B2 | BUY | 50 | 50,90 |
| t-1 | B3 | BUY | 150 | 50,90 |

*Obrázek 5.1: Stav Order Booku v čase t a příchozí příkaz k prodeji*

Po vypořádání příchozího příkazu dojde ke spárování se třemi příkazy v Order Booku. Dva z nich (nejstarší dva) jsou vypořádány zcela, a tedy jsou odstraněny z Order Booku. Poslední z příkazů stále zůstává v Order Booku, i když již částečně vypořádán. Tento stav znázorňuje následující situace (Obrázek 5.2).

| Čas. razítko | Příkaz # | BUY/SELL | Vypořádané loty | Cena |
|---|---|---|---|---|
| t-3 | B1 | FILL | 100 | 50,90 |
| t-2 | B2 | FILL | 50 | 50,90 |
| t-1 | B3 | FILL | 50 | 50,90 |

**Order Book**

| Čas. razítko | Příkaz # | BUY/SELL | Zbývající loty | Cena |
|---|---|---|---|---|
| t-1 | B3 | BUY | 100 | 50,90 |

*Obrázek 5.2: Znázornění FIFO algoritmu a stav Order Booku po vypořádání příchozího příkazu*

# 5.3 Pro-Rata

Pro-Rata je algoritmus založený na poměru. Jeho cílem je rozdělit příchozí příkaz mezi protější příkazy na stejné cenové úrovni. Alokované množství každého takto párovaného příkazu se rovná jeho podílu na dané cenové úrovni trhu.
Existuje celá řada variací Pro-Rata algoritmu. Já se zde pokusím vysvětlit takzvanou „vanilla“ variantu. Příkazy jsou vypořádávány na základě vzorce:

$$A_n = \frac{v_n}{\sum_{r=1}^{N} v_r} * L \tag{5.1}$$

, kde
$A_n$ – alokace příkazu $n$ pro nákup či prodej v Order Booku
$v_n$ – počet lotů příkazu $n$ Order Booku
$N$ – celkový počet příkazů pro prodej či nákup na dané cenové hladině v Order Booku
$L$ – počet lotů příchozího příkazu pro nákup či prodej

Pro demonstraci využiji stejnou výchozí situaci Order Booku jako v případě algoritmu FIFO (viz Obrázek 5.1). Při vypořádání příchozího příkazu dojde ke spárování se třemi příkazy v Order Booku. Distribuce alokací je zde však odlišná – všechny tři příkazy pro nákup stále zůstávají v Order Booku, dále již částečně vypořádané. [14]

| Čas. razítko | Příkaz # | BUY/SELL | Vypořádané loty | Cena |
|---|---|---|---|---|
| t-3 | B1 | FILL | (100/300)*200 = 67 | 50,90 |
| t-2 | B2 | FILL | (50/300)*200 = 33 | 50,90 |
| t-1 | B3 | FILL | (150/300)*200 = 100 | 50,90 |

**Order Book**

| Čas. razítko | Příkaz # | BUY/SELL | Zbývající loty | Cena |
|---|---|---|---|---|
| t-3 | B1 | BUY | 33 | 50,90 |
| t-2 | B2 | BUY | 17 | 50,90 |
| t-1 | B3 | BUY | 50 | 50,90 |

*Obrázek 5.3: Znázornění Pro-Rata algoritmu a stav Order Booku po vypořádání příchozího příkazu*

# 6 Komunikační (transportní) vrstva

Jak již bylo řečeno v kapitole o multiagentních systémech (4), podstatnou úlohou prostředí je zajištění komunikace mezi agenty. Agenti spolu komunikují přímo (peer-to-peer), či nepřímo (zde prostřednictvím Order Booku a pohybu cen).
Následující kapitola proto stručně popisuje knihovnu ZeroMQ, kterou jsem si vybral pro implementaci transportní vrstvy vytvářené platformy.

## 6.1 Základní charakteristika ZeroMQ

ZeroMQ (také známý jako ZMQ nebo 0MQ) je „concurrency light-weight framework“, který se zpravidla používá pro distribuované systémy bez tzv. *Message Brokera* – centrální prvek, přes který proudí veškerá komunikace v systému. Komunikace bez Message Brokera by měla být rychlejší. Na druhou stranu to znamená komplikovanější návrh a konfiguraci komunikační vrstvy, je třeba dávat větší pozor na souběh jednotlivých částí systému. Rychlost je však pro nás prioritou.
Dalším neméně důležitým pozitivem knihovny ZMQ je jeho množství existujících portací v různých programovacích jazycích. Přesněji řečeno se jedná o wrappery napsané v konkrétních jazycích kolem jádra ZMQ napsaného v jazyce C.

Vlastnosti ZMQ:

- Asynchronní I/O handling, a tedy neblokující komunikační rozhraní => aplikace nepotřebují žádné zámky, semafory a jiné blokující mechanismy.
- Jednotlivé komponenty se mohou připojovat a odpojovat dynamicky, žádné typické paradigma typu „nejprve server, pak klient“.
- Automatické řazení zpráv do front, když je potřeba; ZMQ přináší i postupy jak řešit možné přeplnění front (tzv. „high-water mark“).
- Jednotlivé komponenty spolu mohou komunikovat přes libovolný transportní protokol (TCP, multicast, in-process, inter-process) bez nutnosti změny aplikačního kódu.
- Bezpečně řeší blokující (zpomalující) příjemce zpráv podle zvolené strategie (návrhového vzoru – viz dále).
- Nabízí celou škálu použitelných návrhových vzorů a topologií (Request-Reply, Publisher-Subscriber, aj.).
- Možnost vytváření proxy pro zjednodušení síťové komunikace.
- Jednoduchý a spolehlivý framing dat.
- Nediktuje přesný formát přenášených dat, data mohou být libovolné velikosti.
- Snaží se o „inteligentní“ error handling.

Z výše napsaného vyplývá, že ZMQ se stará i o síťové připojení, které však plně zastřešuje, odstiňuje nás od problémů typu dočasné odpojení klienta, kdy se automaticky postará o znovu připojení a my se ani nic nedozvíme. Tím pádem se můžeme zaměřit na to, co nás opravdu zajímá – masivní paralelizaci procesů. V tomto je ZMQ silné. Na druhou stranu nám nedává do rukou hotové řešení, jsou to spíše jen části skládačky, které si musíme poskládat sami podle svých preferencí a potřeb.
ZMQ na rozdíl od klasického připojení typu socket umožňuje připojit jeden socket k více adresám (tzv. *endpoint*). ZMQ si definuje svůj vlastní socket, jehož chování se pak řídí typem socketu (REQ, REP, PUB, SUB, …). Na rozdíl od TCP socketu ZMQ socket může být mapován na vlákna, procesy, ale také na TCP spojení.

## 6.2 Návrhové vzory

Jednotlivé zprávy mezi sockety jsou distribuovány podle určitých návrhových vzorů (tzv. *patterns*). Základními návrhovými vzory v ZMQ jsou:

- *Request-reply*, který slouží k propojení klientů se službami, používá se pro vzdálené volání procedur (RPC) a distribuci úkolů.
- *Publish-subscribe*, který propojuje tzv. *publishery* (strana, která rozesílá nějaká data) s tzv. *subscribery* (strana, která tato data odebírá). Je to tedy návrhový vzor pro distribuci dat.
- *Pipeline*, který propojuje jednotlivé uzly do schématu umožňující distribuci paralelních úkolů.
- *Exclusive pair*, který propojuje 2 sockety exkluzivně. Používá se pro spojení dvou vláken do jednoho procesu.

Tyto návrhové vzory jsou implementovány vždy v páru shodujícími se sockety:

- PUB a SUB
- REQ a REP
- REQ a ROUTER
- DEALER a REP
- DEALER a ROUTER
- DEALER a DEALER
- ROUTER a ROUTER
- PUSH a PULL
- PAIR a PAIR

Jiné kombinace socketů nejsou podporovány a není zaručeno, jak se takové spojení bude chovat.
Nad základními návrhovými vzory, které jsou v ZMQ přímo implementovány, se budují návrhové vzory vyšší úrovně, které se pak dají již přímo aplikovat v reálných nasazeních. Mezi takové návrhové vzory

patří i spolehlivé (*reliable*) Request-Reply návrhové vzory, které nám zaručují určitou úroveň kontroly nad doručováním zpráv při chybových stavech. Právě těmito chybovými stavy je v ZMQ spolehlivost definována. Možné příčiny chybových stavů v distribuovaných ZMQ aplikacích jsou seřazeny podle jejich pravděpodobnosti [15]:

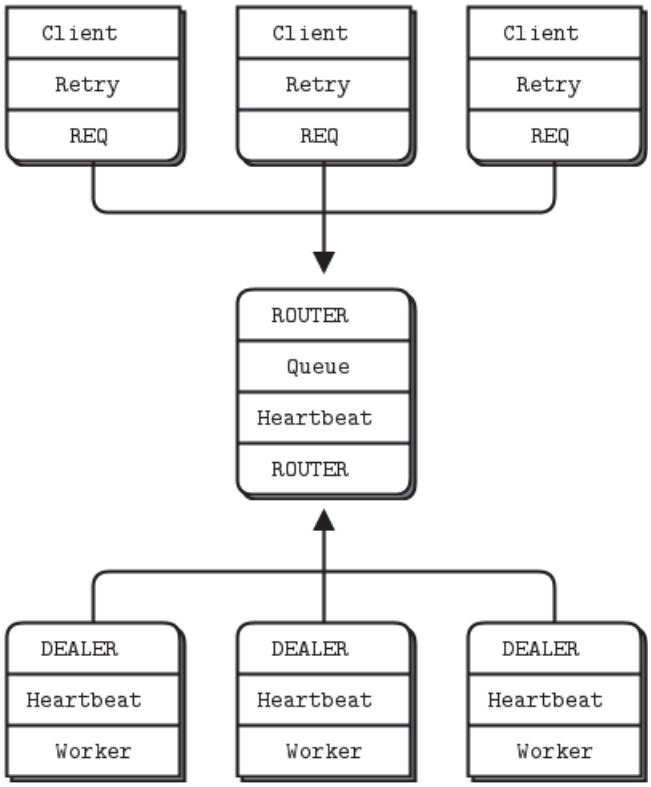
1. Chyba v aplikačním kódu. Ten může havarovat, zamrznout a přestat odpovídat na požadavky, špatná alokace paměti atd.
2. Chyba v systémovém kódu, která poté způsobuje podobné problémy jako výše uvedená chyba v aplikačním kódu. Systémový kód je však zpravidla více spolehlivý než aplikační kód. Příkladem je přetečení paměti kvůli zapisování zpráv do interní fronty pro pomalejší klienty.
3. Fronty zpráv mohou přetéci a systém začne tyto zprávy mazat. Tím pádem přicházíme o data, na nichž však fungování systému může být závislé. Příkladem je systém, který se již naučil brutálně vypořádávat s pomalými klienty – mazat data, která by způsobovala chybu v paměti.
4. Chyba v síťovém připojení typu část systému se dostane mimo dosah WiFi sítě a podobně. ZMQ sice řeší opětovné připojení v takových případech, avšak v mezičase již mohlo dojít ke ztrátě dat.
5. Chyba hardware.
6. Síťové připojení může padnout z jiných příčin jako například poškození portů, na kterých je běžící aplikace závislá.
7. Živelné pohromy, problém z chlazením systému apod.

Zmíněné spolehlivé návrhové vzory řeší prvních 5 příčin, které jsou však zároveň příčinami nejčastějšími. My si zde blíže popíšeme návrhový vzor *Paranoid Pirate*, který je v diplomové práci také implementován a který je právě jedním ze spolehlivých návrhových vzorů pro schéma zapojení Request-Reply.

> *... for systems belonging to the singular part of the stability boundary a small change of the parameters is more likely to send the system into the unstable region than into the stable region. This is a manifestation of a general principle stating that all good things (e.g. stability) are more fragile than bad things. It seems that in good situations a number of requirements must hold simultaneously, while to call a situation bad even one failure suffices.*[16]

### 6.2.1 Paranoid Pirate Pattern

Tento robustní návrhový vzor má 2 obrovské výhody oproti klasickému schématu *Request-Reply*. Poskytuje takzvaný *load-balancing*, tedy rozdělení práce (požadavků) mezi více workerů (serverů), a umožňuje také detekci nevyřízených požadavků. A to ať už vlivem chyby na straně serveru, anebo na straně fronty zpráv samotné (kdy v případě jednoduššího návrhového vzoru *Simple Pirate* je právě fronta slabým článkem schématu a nelze se z chyby v ní zotavit).




*Obrázek 6.1: Návrhový vzor Paranoid Pirate* [15]

Na Obrázek 6.1 můžeme vidět schéma zapojení pro návrhový vzor Paranoid Pirate. Ten se skládá z následujících částí:

**Client** – jakýkoliv klient, který chce využívat poskytovaných služeb. Pro připojení používá pochopitelně socket typu REQ (Request) a má schopnost detekovat nevyřízený požadavek a popřípadě jej odeslat znovu. Detekce je provedena pomocí jednoduché techniky, kdy se kontroluje, zda-li odpověď dorazí do určité doby od odeslání požadavku (tzv. timeout).

**Queue** – fronta odeslaných požadavků, které se dále přeposílají workerům. První strana fronty je implementována pomocí socketu typu ROUTER a slouží jako připojovací bod pro klienty (*front-end*). Druhá strana fronty slouží ke komunikaci směrem k workerům a je také typu ROUTER (*back-end*).

**Worker** – daný server, který chceme požadavkem z klienta oslovit a který poskytuje určitou službu. Jeho komunikačním prostředkem je socket typu DEALER.

Jak fronta, tak server používají k oznamování činnosti (a tedy i k odhalování problémů) techniku zvanou *heartbeat*. Jednoduše řečeno je to opakující se zasílání krátkých zpráv mezi sebou navzájem s cílem, dát protějšku najevo svou aktivitu. Při určitém počtu takových nedoručených pingů, můžeme reagovat na tuto situaci jako na chybový stav – např. odstranění určitého workera z fronty pro distribuci zpráv.

# 7 Serializace dat

Protože data mezi agenty mohou být posílána po síti, je nutné je nějakým způsobem serializovat. Framework ZMQ žádné specifické řešení nevyžaduje, a proto je plně v kompetenci programátora, kterou cestou se vydá. Existuje mnoho řešení pro serializaci dat, některá jsou více standardní, my však hledáme přístup, který nám umožní, co nejvyšší výkon.
V následující kapitole srovnávám možná řešení pro serializaci dat, nejprve uvedu známé textové formáty XML, JSON, které však následně nahradím řešeními pro binární serializaci dat. Uvedu zde srovnání 3 knihoven pro binární serializaci dat – Protocol Buffers, Apache Thrift a Apache Avro.

## 7.1 XML

Pro předávání dat mezi různými systémy se již dlouhá léta využívá značkovací jazyk XML. Tento jazyk je de facto standardem na poli předávání dat mezi informačními systémy.

```
<?xml version="1.0"?>
<user>
  <name>Robert</name>
  <surname>Saniga</surname>
  <company>Brainpool</company>
</user>
```

*Zdrojový kód 7.1: Ukázka XML struktury*

Například webové služby používají protokol SOAP, jenž je právě aplikací jazyka XML. Jako výhoda tohoto řešení bývá označována vlastnost jazyka XML, a sice to že se jedná o člověkem čitelný formát – laicky řečeno komunikaci mezi systémy využívající protokolu SOAP byste si mohli přečíst. Chyba v komunikaci by pak měla být lehko odhalitelná.

## 7.2 JSON

V poslední době se hodně do popředí dostává také jiný značkovací jazyk – JSON.
Ať už je to vlivem stoupající obliby javascriptových frameworků, nebo jen pouhou potřebou po méně „ukecaném“ řešení než je XML, JSON je stále více využíván pro např. AJAX komunikaci. Nebo také pro jiný typ webových služeb, a sice webových služeb postavených na technice REST (využívání jednoduchých HTTP volání a HTTP metod). JSON je také textovým formátem a komunikace je tedy rovněž člověkem čitelná.

```
{
user: {
      name: Robert,
      surname: Saniga,
      company: Brainpool
      }
}
```

*Zdrojový kód 7.2: Ukázka JSON struktury*

# 7.3 Binární serializace dat

Předcházející dvě řešení mají jednu společnou nevýhodu – jsou to textové formáty. Jsou sice „lehce" čitelné člověkem, to však neplatí pro výpočetní systémy, které tato data musí složitě pársovat[4] a to jim ubírá na výkonu. Rovněž jsou takto přenášená data stále příliš velká ve srovnání s jejich binární reprezentací.
Dokonce i nadcházející verze protokolu HTTP (HTTP/2) bude mít binární podobu. Jako hlavní důvody skupina IETF HTTP Working Group, která stojí za vývojem a specifikací nové podoby HTTP protokolu, uvádí nutnost komplikovaného zpracování bílých znaků, velkých písmen, ukončování řádků, prázdných řádků atd. Například HTTP ve verzi 1.1 údajně kvůli tomu definuje až 4 způsoby pársování zpráv. V nové verzi bude existovat pouze jeden způsob. [17]
V následujících podkapitolách budou podrobněji probrána existující řešení pro binární serializaci dat.

## 7.3.1 Protocol Buffers

Jedná se o jazykově a platformě nezávislý mechanismus serializace dat vyvíjený společností Google. Ti jej sami využívají ve svých webových službách. Nabízí implementaci v celé řadě programovacích jazyků, nové stále přibývají. Oproti XML jsou Protocol Buffers 3x – 10x menší a 20x – 100x rychlejší při zpracování. Co se týče srovnání s JSON, budou čísla podobná. [18]
Strukturovaná data serializovaná pomocí Protocol Buffers se musejí nejprve popsat pomocí definičního jazyka IDL. K tomu slouží soubory s příponou .proto. Definice jsou velice podobné definicím datových položek z programovacích jazyků typu Java. Místo tříd zde pracujeme s takzvanými `message`. Protobuf umožňuje využívat jak skalárních hodnotových typů, tak typů kompozitních, nebo také například enumů (výčtů). Mezi skalární typy patří `double`, `float`, různé typy znaménkových a neznaménkových `integer`ů (s fixní, proměnnou délkou, 32, 64 bitové), `boolean` hodnota, řetězec (`string`) a libovolně dlouhá sekvence bytů (`bytes`). Každá datová položka musí být definovaná současně s jedním z modifikátorů `required` (povinná položka obsažená ve zprávě právě jedenkrát), `optional` (zpráva může obsahovat položku jednou, nebo vůbec), anebo `repeated` (datová položka je obsažena v přenášených datech 0 až n-krát). Každá položka je rovněž povinně definována

[4] zpracovávat do strojem čitelné podoby

s celočíselnou hodnotou, která se používá k mapování hodnoty v binární podobě. Číslo musí být jedinečné v jedné definici typu (`message`). Zpravidla čísla začínají hodnotou 1 a jsou vždy inkrementována o jedničku. Nejprve bývají definovány datové položky vyskytující se ve zprávě četněji následovány položkami méně četnými. Číslo totiž rovněž udává, pomocí kolika bytů bude datová položka kódována ve výsledné binární reprezentaci. Položky s hodnotou 1-15 jsou kódovány právě jedním bytem, položky s hodnotou 16-2047 právě dvěma byty atd.
Výsledná .proto definice se prožene Protocol Buffer kompilátorem (protoc), jenž v případě Javy vygeneruje odpovídající třídy pro každý definovaný message type. Rovněž vygeneruje Builder pro instanciaci těchto tříd. Vše je zabaleno v .java souboru pojmenovaném podle názvu z .proto definice.
Jednotlivé .proto definice mohou být do sebe navzájem importovány a využívány pro další definice, popřípadě rozšiřovány.

```
message EventProto {
  required uint64 id = 1;
  required uint64 timestamp = 2;

  enum EventType {
        Order = 1;
        Fill = 2;
        MarketEvent = 3;
        Position = 4;
        Account = 5;
        RegisterStrategy = 6;
  }

  required EventType event_type = 5;

  extensions 100 to 199; // extensions for concrete types
}
```

*Zdrojový kód 7.3: Ukázka .proto definice*

### 7.3.2 Apache Thrift

Open Apache projekt, interně jej využívá pro své služby Facebook. Umožňuje stejně jako protobuf definovat také procedury, na rozdíl od něj však poskytuje již hotovou implementaci RPC rozhraní (v případě protobuf si jej musí uživatel naimplementovat sám).
V porovnání s protobuf nabízí rovněž bohatší definiční jazyk, jsou zde k dispozici různé typy kolekcí a také větší množství oficiálně podporovaných programovacích jazyků.

```
struct Trade {
  1: required i32 id;
  2: required i64 timestamp;
  3: required double price = 0.00;
}
```

*Zdrojový kód 7.4: Ukázka Apache Thrift definice*

### 7.3.3 Apache Avro

Další open Apache projekt. Na rozdíl od předchozích řešení jsou data serializovaná Avrem vždy posílána také se schématem (popis struktury, tedy něco jako .proto soubor v případě protobuf). To má své výhody (data mohou být zpracována jakýmkoliv programem později bez znalosti definic, který si strukturu načte spolu s daty), na druhou stranu je velikost dat vždy větší.
K popisu struktury slouží JSON definiční soubor. Jedná se také o RPC framework. Nevyžaduje kompilování definicí ani žádné ručně definované ID pro každou datovou položku.

```
{
  "type": "record",
  "name": "Trade",
  "fields" : [
    {"name": "id", "type": "int"},
    {"name": "timestamp", "type": "long"},
    {"name": "price", "type": "double", "default": "0.00"}
  ]
}
```

*Zdrojový kód 7.5: Ukázka Apache Avro definice*

### 7.3.4 Srovnání

Po zvážení všech pro a proti jsem se rozhodl použít v praktické části k serializaci dat knihovnu Protocol Buffers. Protocol Buffers mají velkou podporu komunity a tím pádem lze nalézt spoustu hotových řešení a příkladů, dokumentace je přehledná a kvalitní, produkt je stále vyvíjen (v době psaní diplomové práce vyšla nová major verze 3.0, která nabízí celou řadu novinek a vylepšení). To že za produktem stojí společnost Google je také jistě velké plus.
Podle benchmarků[5] jsou Protocol Buffers menší co do velikosti serializovaných dat, i rychlejší co do rychlosti serializace/deserializace oproti dalším dvěma představeným řešením pro binární serializaci. [19]

[5] testy rychlosti

# 8 Monitorování

Pro monitorování událostí simulační platformy jsem se rozhodl využít logovacího nástroje. Moderní logovací nástroje umožňují zapisovat do různých typů výstupu, od těch nejjednodušších (textové soubory) až po ty složitější (databáze). Mezi nejznámější nástroje pro logování na platformě Java patří Log4j2.

## 8.1 Log4j2

V současné době jeden z nejmodernějších nástrojů pro logování na platformě Java. Přináší značná vylepšení oproti svému předchůdci – Log4j – a zároveň se snaží poučit z chyb jiného známého logovacího nástroje – Logback.

Kromě zmíněné možnosti definice různých výstupních formátů, nabízí Log4j2 také možnost filtrování výstupu na základě kontextových dat, značek, regulárních výrazů a dalších parametrů. Také možnost změny konfigurace v běžícím programu se může hodit - například změna úrovně logování, kdy chceme něco blíže prozkoumat, odladit, lze provést bez zastavení aplikace. Log4j2 je rovněž dostupný pro další API, jako SLF4J a Commons Logging API.

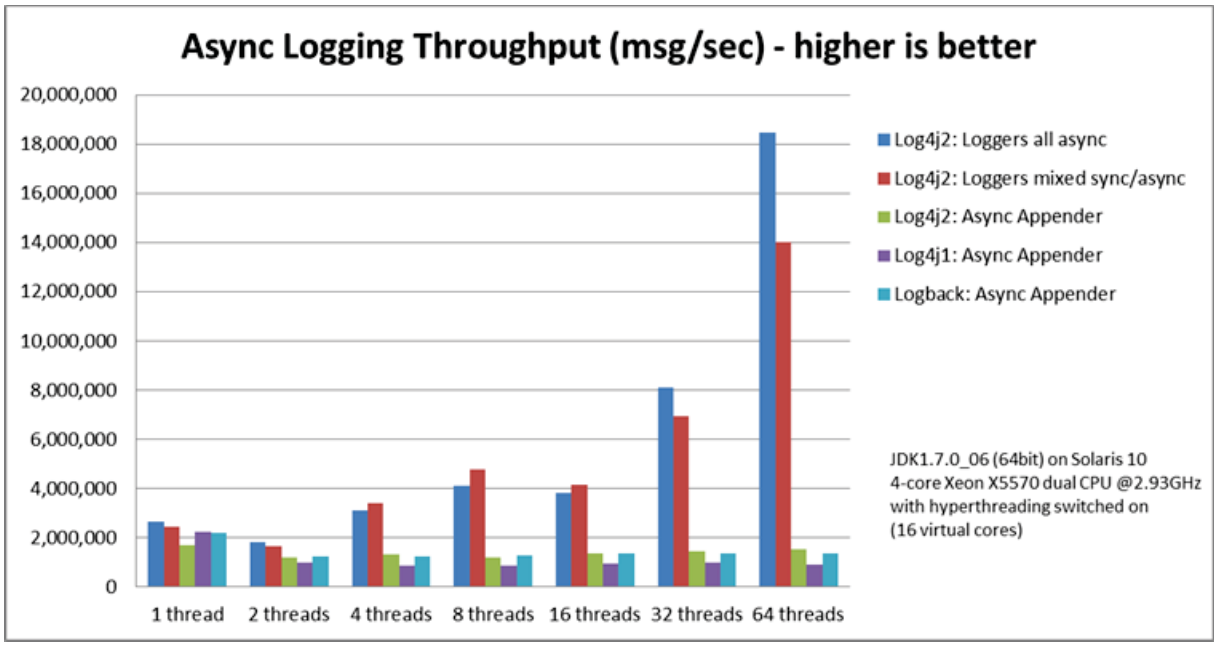
V čem se však Log4j2 snaží být lepší oproti svým předchůdcům, je výkonnost. Díky novým asynchronním loggerům postaveným na knihovně LMAX Disruptor má mít v případě více-vláknových aplikacích 18x vyšší propustnost a nižší latenci než Log4j 1.x a Logback. [20] Podívejme se na srovnání.

### 8.1.1 Srovnání výkonu s ostatními knihovnami

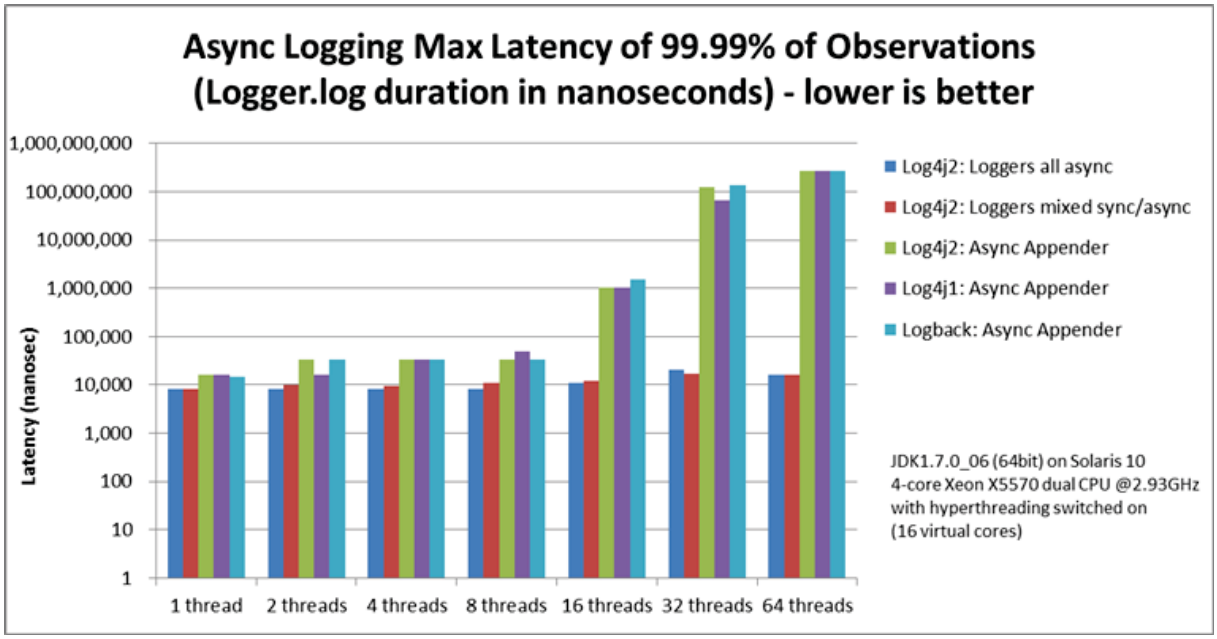
Podle zveřejněných testů (Obrázek 8.1) je patrné, že i když v případě využití pouze jednoho vlákna jsou výsledky coby do výkonnosti jednotlivých knihoven velice podobné, v případě více-vláknových aplikací využívají asynchronní logery z Log4j2 lépe výpočetní jádra počítače a doslova drtí ostatní knihovny rozdílem několika řádů – v případě využití 64 vláken je Log4j2 (288 997 zpráv/s) o přibližně 267 694 zpráv/s rychlejší než Logback (21 303 zpráv/s), což představuje asi o 1156,6% vyšší propustnost [21].

Pro testování latence (zpoždění) je měřeno, jak dlouho trvá návrat z volání metody Logger.log (Obrázek 8.2). Zatímco si aplikace Async Loggeru udržuje maximální zpoždění v 99,99% pozorovaných případů někde mezi 10-20 us pro různý počet dostupných vláken, pro vyšší počet současně běžících vláken je zpoždění v případě Async Appenderů až o 4 řády vyšší (pro 64 současně běžících vláken).

V případě průměrných hodnot zpoždění jsou až do situace s 8 současně běžícími vlákny zpoždění srovnatelné (Async Appendery jsou asi jen 2x-3x pomalejší než Async Logger), v případě vyššího počtu běžících vláken je však rozdíl také již několik řádů.




*Obrázek 8.1: Srovnání propustnosti logovacích knihoven*[6]




*Obrázek 8.2: Srovnání latence logovacích knihoven*[7]

[6] Obrázek převzat z http://logging.apache.org/log4j/2.x/manual/async.html#Asynchronous_Throughput_Comparison_with_Other_Logging_Packages

[7] Obrázek převzat z http://logging.apache.org/log4j/2.x/manual/async.html#Latency

Asynchronní loggery v Log4j2 jsou implementovány pomocí LMAX Disruptor knihovny, což je inter-thread messaging knihovna. Autoři knihovny tvrdí [22]:

> *... using queues to pass data between stages of the system was introducing latency, so we focused on optimising this area. The Disruptor is the result of our research and testing. We found that cache misses at the CPU-level, and locks requiring kernel arbitration are both extremely costly, so we created a framework which has "mechanical sympathy" for the hardware it's running on, and that's lock-free.*

Zastávají tak podobnou filozofii jako autor ZMQ, který je rovněž toho názoru, že pro opravdu rychlou komunikaci mezi jednotlivými vlákny je nutné se vyhnout zámkům, semaforům a jiným zamykacím technikám.

### 8.1.2 Příklad konfigurace

Ukázka konfigurace Log4j2 (Zdrojový kód 8.1) definuje asynchronní logger pro monitorování obchodních událostí platformy. Události jsou logovány do souborů s názvem `events-[pořadové číslo logu].log.gz` ve složce `target`. Logovací soubory jsou komprimovány pomocí knihovny GZIP, stará se o to Log4j2 samotný. Jednotlivé soubory jsou děleny po 50 MB a vždy jen poslední dva soubory jsou k dispozici, starší soubor se smaže.

```
…
  <RollingRandomAccessFile name="EventsFile" immediateFlush="false"
append="false" fileName="target/events.log"
filePattern="target/events-%i.log.gz">
    <PatternLayout>
      <pattern>%d | %m%n</pattern>
    </PatternLayout>
    <Policies>
      <SizeBasedTriggeringPolicy size="50 MB"/>
    </Policies>
    <DefaultRolloverStrategy max="2"/>
  </RollingRandomAccessFile>
</Appenders>
<Loggers>
  <AsyncLogger
name="cz.strattonoakmont.ats.core.monitor.Log4JLoggerAdapter"
level="INFO" additivity="false">
    <AppenderRef ref="EventsFile" />
  </AsyncLogger>
…
```

*Zdrojový kód 8.1: Ukázka konfigurace Log4j2 (z log4j2.xml)*

# 9 Návrh a implementace

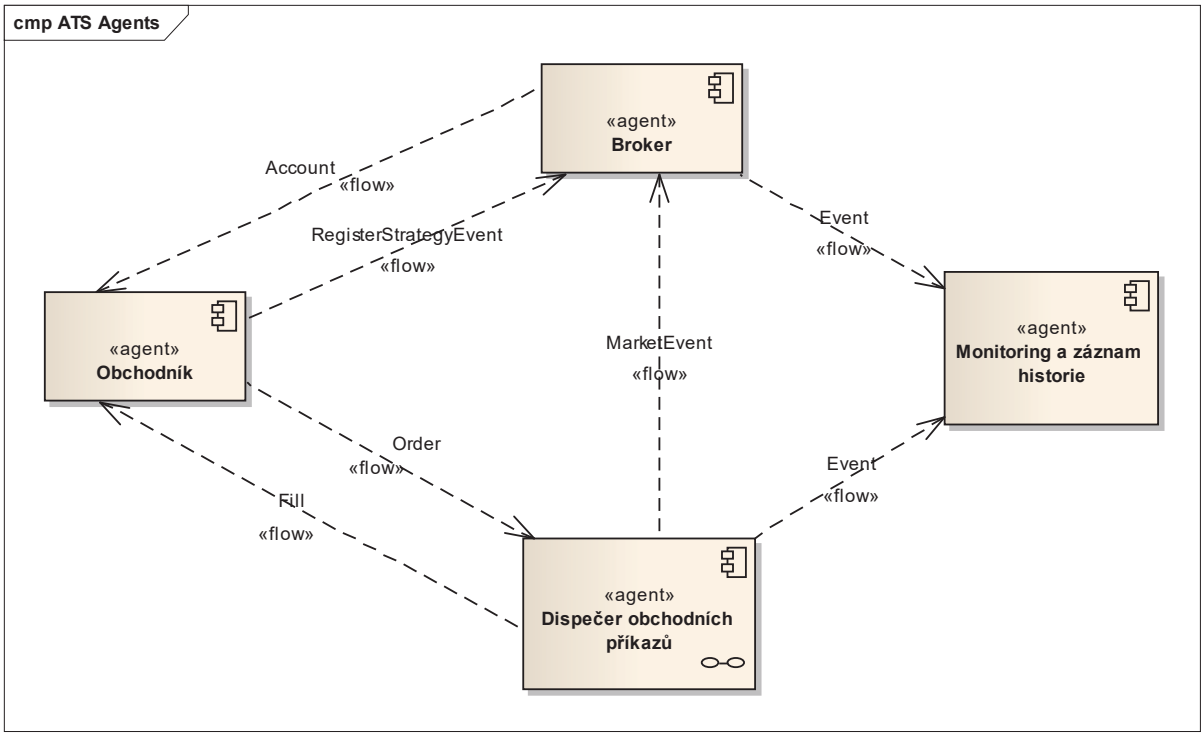
Kapitola popisuje návrh a implementaci simulační platformy. Platforma bude napsána v programovacím jazyce Java, který je objektově orientovaným programovacím jazykem a pro nějž existují portace všech výše uvedených knihoven.

## 9.1 Identifikace jednotlivých částí systému

Z hlediska návrhu systému, můžeme rozlišit několik typů agentů:

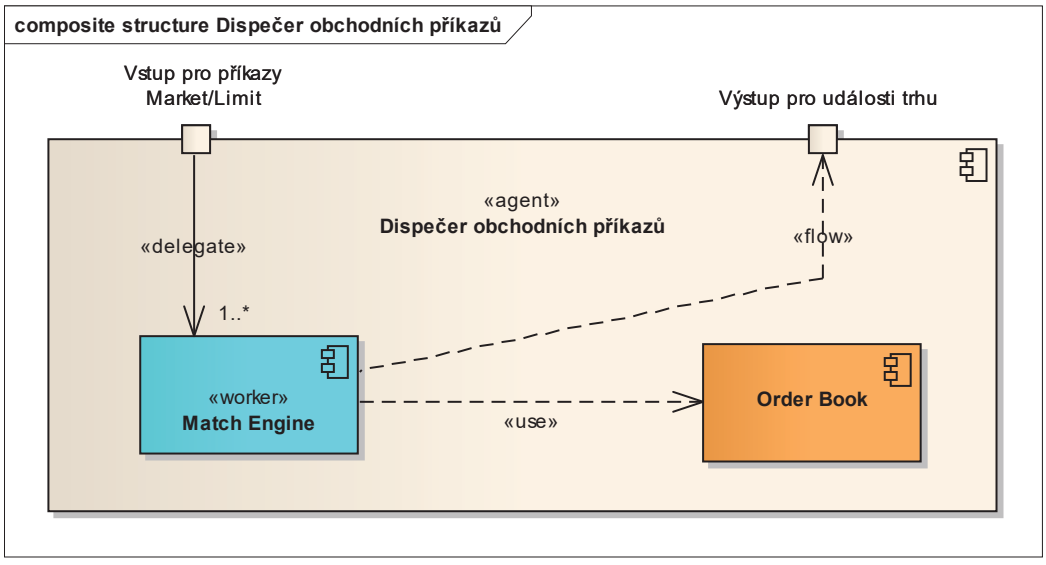
1) Obchodník
2) Broker
3) Dispečer obchodních příkazů (Match Engine a Order Book na pozadí)
4) Agent, který monitoruje události simulační platformy a zaznamenává historii

Vztah mezi těmito agenty může být znázorněn pomocí diagramu níže (Obrázek 10.1). Jednotlivé relace lze nejlépe vyjádřit prostřednictvím dat, které si agenti navzájem vyměňují.




*Obrázek 10.1: Vztah mezi agenty v obchodní platformě*

Celý proces začíná u obchodníka, který brokerovi posílá požadavek na registraci účtu (*RegisterStrategyEvent*). Ten v případě kladného vyřízení posílá zpět informace o zřízeném účtu, jako jsou ID účtu, počáteční zůstatek, výše poplatků za transakci. Ve chvíli, kdy se obchodník rozhodne pro nákup či prodej určitého množství akcií, posílá požadavek (*Order*) k dispečeru obchodních příkazů. V případě příkazu se stanovenou cenou posílá *LimitOrder*, v případě kdy chce prodat či nakoupit ihned, posílá *MarketOrder*. Dispečer potvrzuje přijetí příkazu odesláním ID příkazu zpět obchodníkovi. Ve chvíli, kdy dojde k vypořádání příkazu, ať už částečnému nebo úplnému, posílá dispečer obchodníkovi informace o tomto vypořádání (vypořádané množství akcií, jejich cena, částečné/úplné vypořádání) pomocí události *Fill*. Stejnou událost posílá také brokerovi kvůli počítání do pozice.
V reálném světě zadává obchodník jednotlivé příkazy prostřednictvím brokera. Obchodník pošle brokerovi příkaz typu Market/Limit a ten teprve zadá příkaz na trh. Až dojde k vypořádání příkazu, účtuje si broker za tuto transakci poplatky. Já jsem se rozhodl cestu zadávání příkazů na trh „zkrátit“. Obchodník posílá příkazy přímo do trhu sám, což by se mělo pozitivně promítnout v rychlosti vypořádávání jednotlivých příkazů, potažmo celého navrhovaného systému. Transakční poplatky jsou však stále započítávány. Broker musí jednotlivé obchody sledovat, aby na jejich základě mohl počítat statistiky pro zaregistrované strategie, obchodníky a aby byl schopen zapisovat pohyby a počítat zůstatek na účtu obchodníka.
Všechny události simulační platformy odchytává agent pro monitorování a záznam historie. Ten je napojen na rozhraní brokera a dispečera obchodních příkazů, přes které všechny události platformy proudí.
Dispečer obchodních příkazů - přesněji řečeno Order Book - generuje celou řadu událostí, které nastanou na trhu v různé fázi obchodování. Některé z těchto událostí přijímá právě broker, kvůli výše zmíněným důvodům. Dispečer obchodních příkazů může být detailněji popsán pomocí následujícího diagramu (Obrázek 10.2).

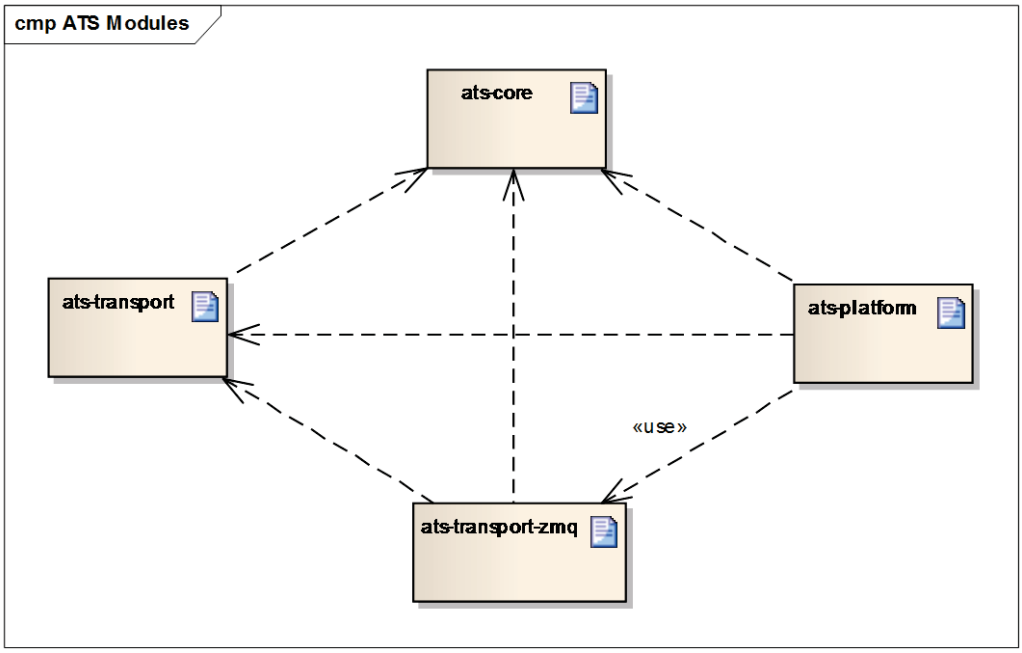



*Obrázek 10.2: Detailní pohled na agenta pro vyřizování obchodních příkazů*

Hlavními částmi této komponenty jsou dříve rozebírané základní prvky akciového trhu - Order Book (kapitola 3.2) a Match Engine (kapitola 5).

# 9.2 Základní struktura platformy

Při implementaci architektury projektu jsem se řídil postupy doporučovanými pro vývoj aplikací pomocí platformy Maven. Apache Maven je moderní buildovací nástroj a nástroj pro komplexní softwarový management. Každý ucelený kus kódu (knihovna, plugin, …) zabalený jako Maven projekt představuje modul. Moduly je potom možné do sebe skládat, a touto cestou je využívat (tomuto mechanismu se říká *dependency*).
Projekt obsahuje jeden tzv. rodičovský (*parent*) modul a pod ním jsou definovány jednotlivé submoduly, které odpovídají určitému účelu, funkcionalitě. Parent modul slouží převážně pro definici společných meta-dat využívaných Mavenem. Sem se umísťují všechny použité knihovny a pluginy v projektu. Je to zároveň jediné místo, kde bychom měli definovat verze těchto knihoven/pluginů. Z pohledu architektury vypadá aplikace následovně:




*Obrázek 10.3: Architektura aplikace a závislost mezi moduly*

Aplikace je rozdělena do 4 modulů:

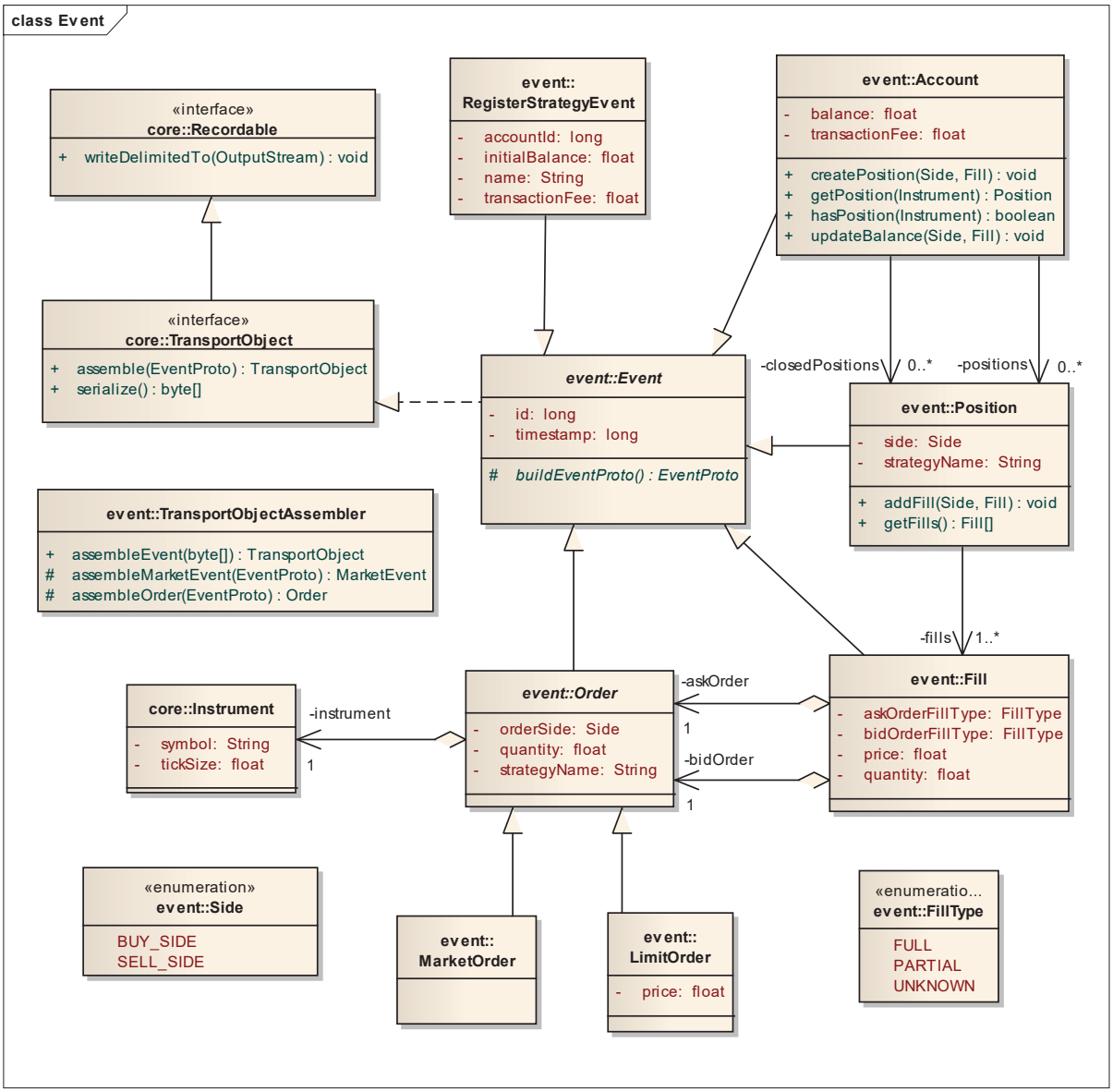
1) **ats-core** – „jádro“ aplikace zahrnující model a protobuf definice; rovněž je zde implementována serializace obchodních dat a událostí pomocí knihovny protobuf
2) **ats-transport** – obsahuje základní rozhraní a definice transportní vrstvy
3) **ats-transport-zmq** – konkrétní (a doposud jediná) implementace modulu ats-transport pomocí knihovny ZMQ; ta je dále využívána samotnou simulační platformou pro komunikaci jednotlivých částí systému mezi sebou navzájem

4) **ats-platform** – sdružuje logiku simulační platformy a definuje jednotlivé základní prvky systému (agenti, Order Book, …)

Dále se podíváme trochu podrobněji na jednotlivé moduly.

# 9.3 Modul ATS core

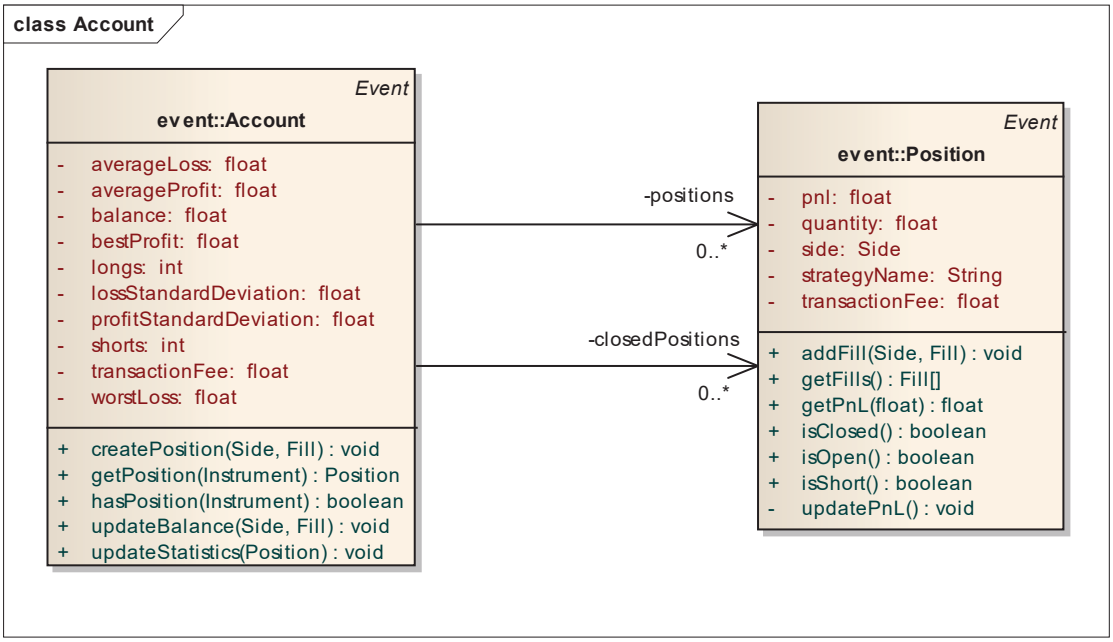
V modulu `ats-core` jsou umístěny všechny datové struktury platformy určené pro výměnu dat. Celá platforma je postavena na komunikaci mezi jednotlivými agenty. Ta je zprostředkována událostmi modelovanými abstraktní třídou `Event`. `Event` je tedy základním kamenem pro výměnu informací. Třída `Event` a její hlavní podtypy jsou znázorněny v následujícím diagramu (Obrázek 10.4).




*Obrázek 10.4: Třídní diagram základních částí modulu ATS core*

Třída `Event` rovněž implementuje rozhraní `TransportObject`, na kterém je definováno, co vyžaduje transportní vrstva – schopnost serializace (metoda `serialize()`) a deserializace (metoda `assemble(EventProto)`). Každý potomek třídy `Event` tedy musí povinně implementovat způsob, jakým se serializuje pro posílání po síti, a také deserializuje ve chvíli, kdy dorazí na místo určení. S deserializací pomáhá třída `TransportObjectAssembler`, která však jen v podstatě na základě několika prvních bajtů rozpozná, o který typ události se jedná, vytvoří jeho instanci a nadále již deleguje práci na tuto instanci konkrétní události. Ostatně ta „ví" nejlépe, které datové položky - proměnné a jejich datový typ - obsahuje. Každý potomek abstraktní třídy `Event` musí rovněž implementovat abstraktní metodu `buildEventProto()`, která ještě dále zjednodušuje (pomocí dědičnosti) způsob, jakým je instance znovu vytvořena ze své serializované formy.
Dalším bodem, který musí platforma splňovat, je schopnost ukládat uskutečněné události systému pro účely dalších analýz. A proto rozhraní `TransportObject` rozšiřuje dále rozhraní `Recordable` (tedy schopnost záznamu), které svým implementacím nařizuje implementovat jedinou metodu – `writeDelimitedTo(OutputStream)`. Při realizaci metody využijeme opět knihovny protobuf (jde především o serializaci dat), která možnost zápisu do výstupního streamu nad svými generovanými třídami `-Proto` již implementuje. Vše je automaticky vygenerováno překladačem protoc, a poté k dispozici vývojáři. Výhoda serializace do binárního formátu je zde patrná. Velikost souborů, do nichž je záznam trhu pořizován a kdy hovoříme o tisících záznamech za vteřinu, je řádově nižší v porovnání s formáty typu XML, JSON.
V diagramu se nachází také dříve zmíněná událost, třída `RegisterStrategyEvent`. Ta poskytuje obchodním strategiím možnost vytvořit u agenta typu broker účet pro počítání zůstatku po obchodních transakcích a zaznamenávání statistik, z jejichž vyhodnocování pak strategie může dále čerpat. Registrace účtu a jeho svázání s příslušnou strategií je nicméně povinností každé strategie a bez ní strategii není umožněno obchodovat. O tom jak jsou strategie implementovány, si však povíme až později, implementace strategií je umístěna v modulu `ats-platform`.
Ve chvíli, kdy je strategie zaregistrována a propojena s účtem, jsou obchodní příkazy realizovány událostí typu `Order`. Přesněji řečeno typu `MarketOrder` nebo `LimitOrder`, které odpovídají příkazům typu Market a Limit (viz kapitola 3.1). Každý obchodní příkaz je identifikován jednak názvem strategie, která jej realizovala, a jednak obchodním instrumentem (třída `Instrument`), kterého se obchod týká. Na těchto dvou identifikátorech závisí následné zpracování příkazu na trhu, konkrétně přiřazení odpovídajícímu Order Booku podle instrumentu a postup při jeho vyřizování. U příkazu, který čeká na vypořádání v Order Booku a který obsahuje stejný identifikátor strategie jako příchozí příkaz protější strany, dochází k rušení akcií odpovídajícímu množství akcií definované položkou `quantity` u příchozího příkazu.
Třída `Instrument` nese kromě jednoznačné identifikace aktiva pomocí tzv. symbolu také hodnotu s názvem `tickSize`. Ta definuje nejmenší možný rozdíl cen v Order Booku. Jinými slovy řečeno, o kolik může být cena aktiva zvyšována, popř. snižována.
Ve chvíli, kdy dojde k vypořádání jakéhokoliv množství akcií, je v Match Engine vytvořena událost typu `Fill`, která je následně odeslána na výstup obchodních událostí. Událost `Fill` obsahuje údaje o obou stranách obchodu, jestli došlo k částečnému, nebo úplnému vypořádání, cenu a množství.
Další velice důležitou částí modelu je třída `Account`. Ta je jednak využívána k držení stavu a počítání určitých statistik jednotlivých účtů, které odpovídají úspěšnosti strategie coby majitele účtu, ale také

k posílání těchto informací dále odpovídajícím obchodníkům. Proto rovněž tato třída dědí z předka všech událostí - ze třídy `Event`. Detailněji popisuje strukturu třídy `Account` schéma na obrázku 10.5.

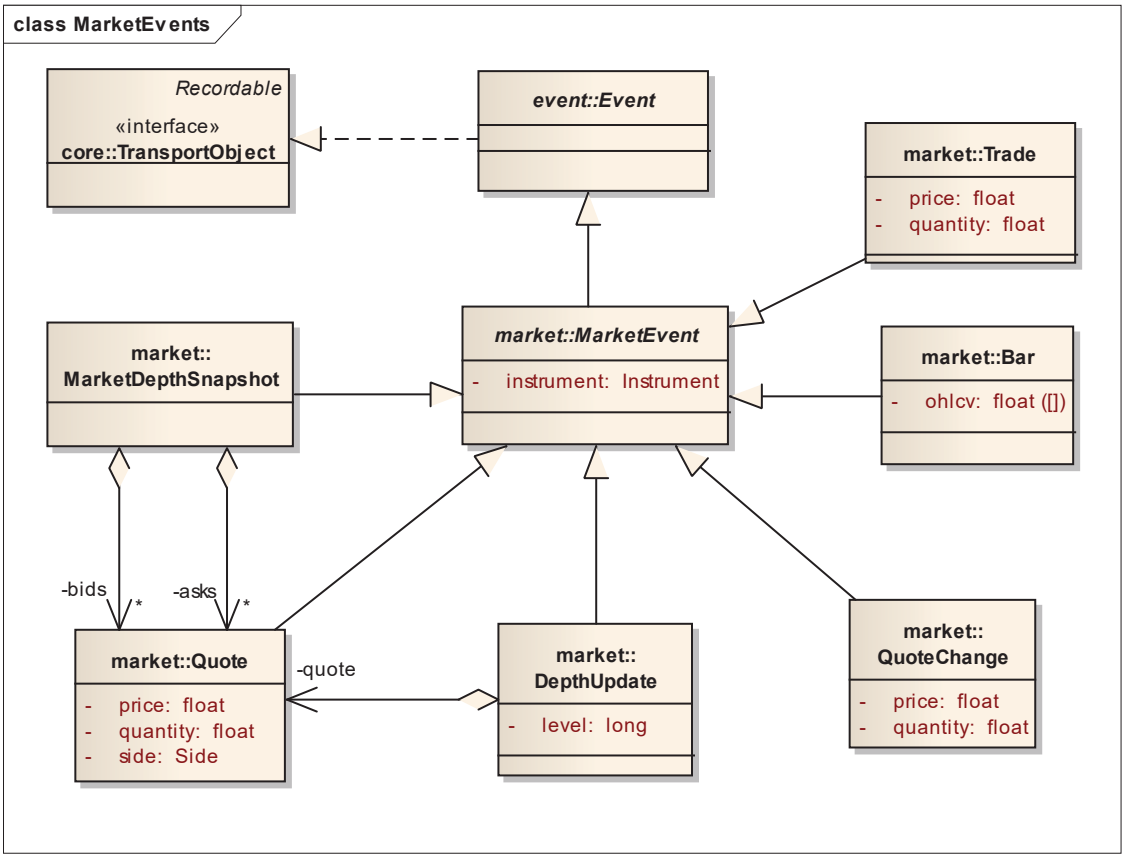



*Obrázek 10.5: Detailní diagram třídy Account a Position a jejich vzájemný vztah*

Každá událost typu `Fill` se počítá do takzvané pozice zúčastněné strategie (obchodníka), v systému modelované třídou `Position`. Pozice může zůstávat otevřená po dobu více vypořádání, jelikož obchodník smí dokupovat/doprodávat akcie do otevřené pozice. Uzavírá se až ve chvíli, kdy množství nakoupených/prodaných akcií je kompletně prodáno/nakoupeno. Podle prvního, otevíracího příkazu je určen typ pozice – dlouhá vs. krátká (atribut `side`). Při dlouhé pozici je otevíracím příkazem nákup, při krátké prodej (podrobněji popsáno v kapitole 3). Pro každou pozici je rovněž počítána hodnota *Profit and loss* (`pnl`), která signalizuje, zda-li a jak moc je pozice zisková či ztrátová.
Na základě pozic jsou ve třídě `Account` počítány statistiky k účtu, potažmo strategii, která je přímo s účtem svázaná. Počítá se zde celkový počet krátkých, dlouhých pozic (`shorts`, `longs`), průměrné hodnoty zisku a ztráty (`averageProfit`, `averageLoss`), nejvyšší zisk (`bestProfit`), nejvyšší ztráta (`worstLoss`), směrodatné odchylky zisku a ztráty (`profitStandardDeviation`, `lossStandardDeviation`). V neposlední řadě je počítán zůstatek účtu (`balance`).

### 9.3.1 Market Event

Speciální podskupinou událostí jsou podtypy třídy `MarketEvent`. Tyto události jsou generovány samotným trhem vykonáváním příkazů v Order Booku a slouží pro výpočty obchodních statistik a také pro takzvanou technickou analýzu. Události jsou popsány v diagramu (Obrázek 10.6) dále.




*Obrázek 10.6: Abstraktní třída MarketEvent a její konkrétní podtypy*

Protože jsou jednotlivé typy událostí definovány jako podtypy třídy `Event` (popř. `MarketEvent`), bylo by dobré tuto dědičnost nějak zachytit i prostřednictvím definicí pro serializaci v protobuf. V protobuf lze dědičnost implementovat pomocí *extensions*. Ukázka takové definice pro událost `Trade` je znázorněna na příkladu (Zdrojový kód 10.1) dále.

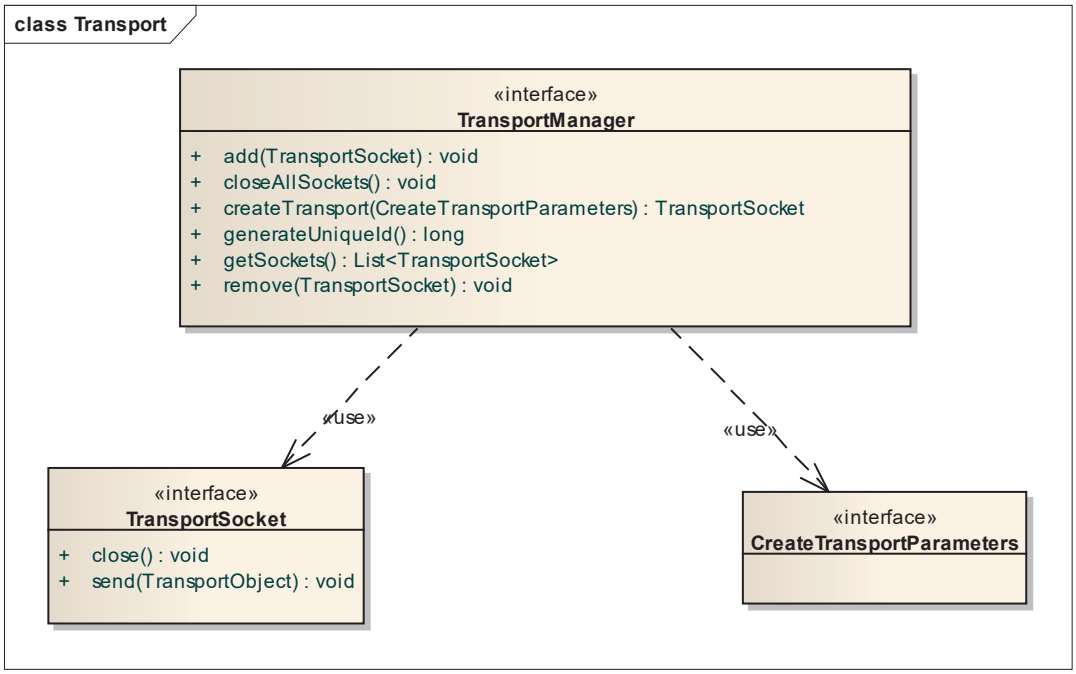
```
message TradeProto {
      extend MarketEventProto {
            required TradeProto event = 300;
      }

      required float price = 1;
      required float quantity = 2;
}
```

*Zdrojový kód 10.1: Ukázka implementace dědičnosti v protobuf*

V tomto příkladu typ `TradeProto` dědí z obecného typu `MarketEventProto`. Z toho důvodu definuje speciální rozšíření s hodnotou 300, což znamená, že na datové položce s hodnotou 300 se bude mapovat podtyp `TradeProto`. Navíc typ (zde vlastně message) `TradeProto` definuje další specifické datové položky – `price` a `quantity`.

# 9.4 Modul ATS transport

V modulu `ats-transport` je definována pouze abstrakce transportní vrstvy. Modul obsahuje sadu základních rozhraní pro práci nad transportní vrstvou, jak je naznačeno na následujícím diagramu (Obrázek 10.7).

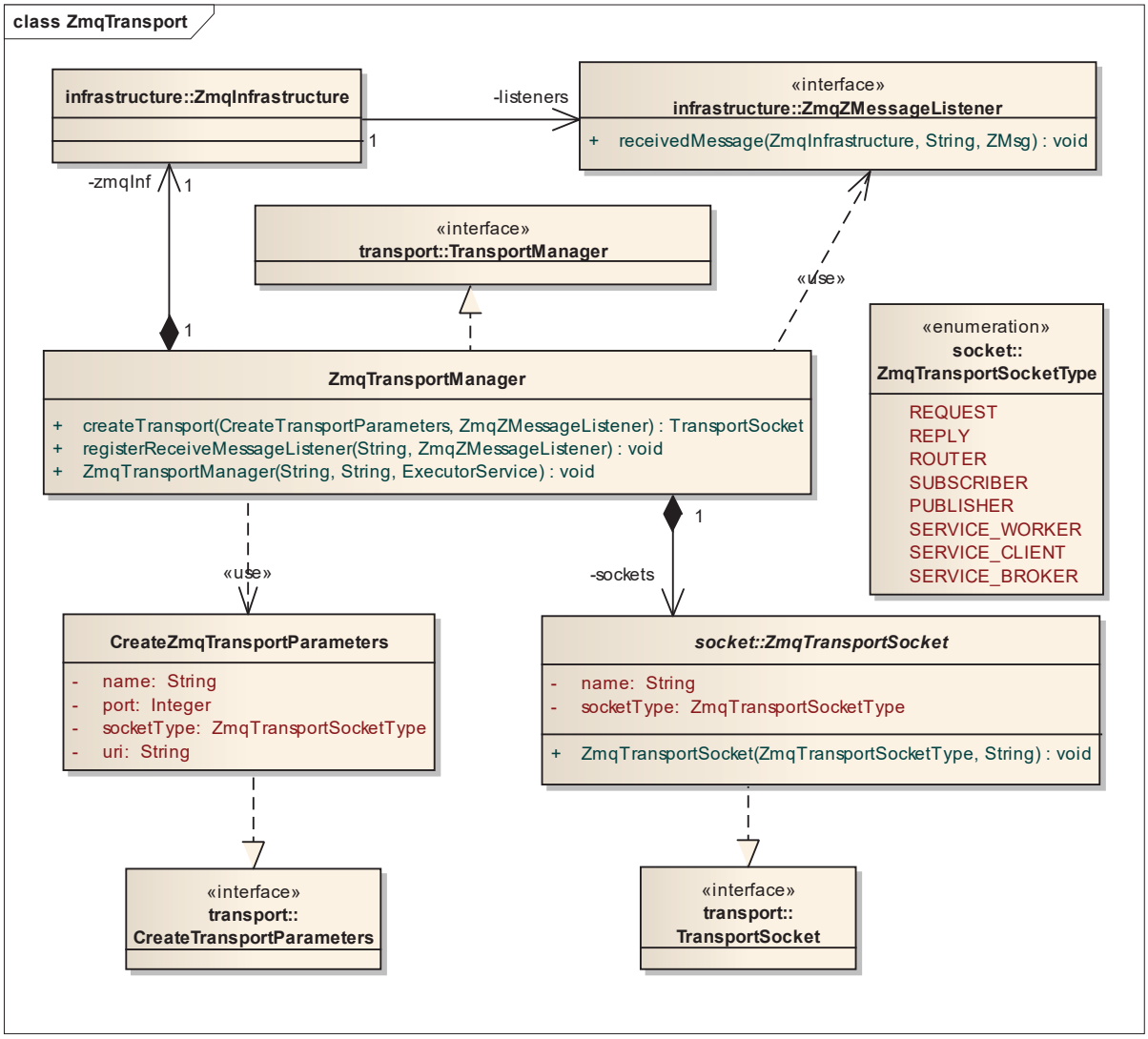



*Obrázek 10.7: Třídní diagram modulu ATS Transport*

Základem je rozhraní `TransportManager`, které se stará o vytváření spojení mezi agenty (metoda `createTransport(CreateTransportParameters)`) a zastřešuje celkovou správu nad sockety těchto spojení. Rozhraní `CreateTransportParameters` zde slouží pouze jako placeholder pro specifické implementace parametrů socketového spojení. Ty pak budou definovat již konkrétní vlastnosti, důležité pro vytvoření konkrétních socketů.
`TransportManager` definuje ještě jednu důležitou operaci – `generateUniqueId()`. Ta má za úkol – jak již sám název napovídá – generovat unikátní ID. Toto ID se pak používá například jako identifikátor jednotlivých událostí. Z toho důvodu musí být zajištěno, že ID bude unikátní v celém systému – napříč několika fyzickými stroji, JVM, v síti – a nejen na jednom konkrétním počítači.
Každý komunikační socket musí implementovat rozhraní `TransportSocket`, které nad ním definuje dvě základní operace – `send(TransportObject)` pro odesílání transportních dat (událostí) a `close()` pro uzavření socketu a uvolnění systémových zdrojů s ním spojených.

# 9.5 Modul ATS transport ZMQ

Modul `ats-transport-zmq` je konkrétní implementací výše uvedeného rozhraní pro komunikaci. Využívá k tomu knihovnu ZMQ. Základní části modulu jsou zachyceny na následujícím diagramu (Obrázek 10.8).




*Obrázek 10.8: Třídní diagram základních částí modulu ATS transport ZMQ*

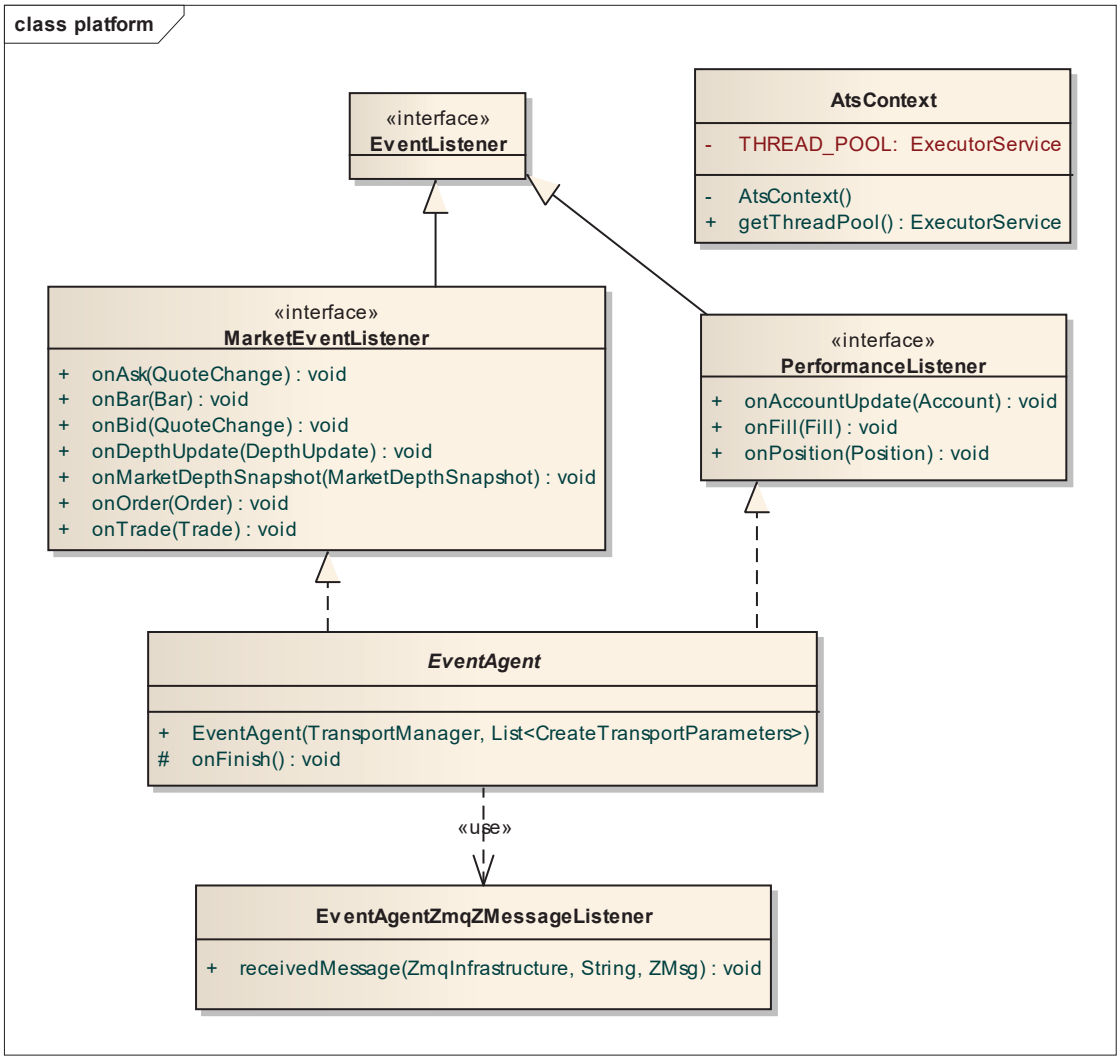
Základní a nejdůležitější částí tohoto modulu z pohledu uživatele simulační platformy ATS je třída `ZmqTransportManager`. Ta implementuje rozhraní `TransportManager` a navíc přidává operace specifické pro komunikaci postavenou na knihovně ZMQ. K tomuto účelu využívá instanci třídy `ZmqInfrastructure`, která zapouzdřuje nízkoúrovňové operace knihovny ZMQ a dále usnadňuje práci s tímto frameworkem. `ZmqInfrastructure` se dále stará o distribuci úkolů nad sockety mezi dostupná vlákna systému, předávání posílaných a přijímaných zpráv cílovým ZMQ

socketům, management ZMQ socketů a poskytuje možnost asynchronního přijímání zpráv, které je definováno rozhraním `ZmqZMessageListener`. Implementací jeho jediné callback metody `receivedMessage(ZmqInfrastructure,String,ZMsg)` a následnou registrací instance tohoto listeneru volání metody `registerReceiveMessageListener(String socket,ZmqZMessageListener)` dochází současně k registraci asynchronního příjmu zpráv na socketu identifikovaného zvoleným názvem. Stejného efektu se současným vytvořením komunikačního socketu dosáhneme vyvoláním metody `createTransport(CreateTransportParameters,ZmqZMessageListener)`.
K vytváření transportního socketu se pro nastavení parametrů (jednoznačný název, typ, uri a nepovinně port) používá objekt typu `CreateZmqTransportParameters` implementující již naznačené rozhraní `CreateTransportParameters`. Existují ještě podtypy této třídy, které nejsou zachyceny na předchozím diagramu z důvodu přehlednosti. Tyto podtypy nesou navíc některé specifické vlastnosti a jsou logicky použitelné jen pro určitý typ socketu – např. název služby ve třídě `CreateZmqServiceTransportParameters`, jejíž instance se používají zpravidla jen a pouze pro vytváření socketu typu `SERVICE_WORKER`. Výčet podporovaných typů socketu je uveden ve třídě typu `enum ZmqTransportSocketType`.
Na základě typu socketu a jeho jedinečného pojmenování se vytváří různé instance podtypů abstraktní třídy `ZmqTransportSocket` reprezentující socket s jeho základními vlastnostmi a schopnostmi – metody z jim implementovaného rozhraní `TransportSocket`. Například socket typu `ZmqSub` (`SUBSCRIBER`) logicky neumožňuje zasílání žádných událostí. Slouží jen k odebírání událostí posílaných socketem typu `ZmqPub` (`PUBLISHER`), a tudíž při vyvolání zde nepodporované metody `send(TransportObject)` vyhazuje vyjímku `UnsupportedOperationException`.

# 9.6 Modul ATS platform

využívá elementární části platformy definované a implementované v předchozích modulech a skládá je do ucelených prvků - agentů. Všechny typy agentů a jejich základní struktury jsou implementovány právě v modulu `ats-platform`.
Z důvodu zjednodušení a sjednocení práce s komunikačním rozhraním byla vytvořena abstraktní třída `EventAgent` (jak je naznačeno na obrázku 10.9) implementující posluchače jednotlivých skupin událostí (`MarketEventListener` a `PerformanceListener`). Pro asynchronní příjem zpráv je implementováno rozhraní `ZmqZMessageListener` (z modulu `ats-transport-zmq`) v privátní třídě `EventAgentZmqZMessageListener`. Tento posluchač odchytává jednotlivé události všech zaregistrovaných transportních rozhraní a ukládá je do interní fronty, odkud se pak předávají do cílových metod typu `onAsk(QuoteChange)`, `onBar(Bar)`, … .
Singletone třída `AtsContext` poskytuje prvkům systému konkrétní implementaci standardního J2SE[8] rozhraní `ExecutorService` (z balíku `java.util.concurrent`) pro řízení životního cyklu a distribuci vláken.

[8] Java Standard Edition – sada základních knihoven platformy Java pro desktopové použití



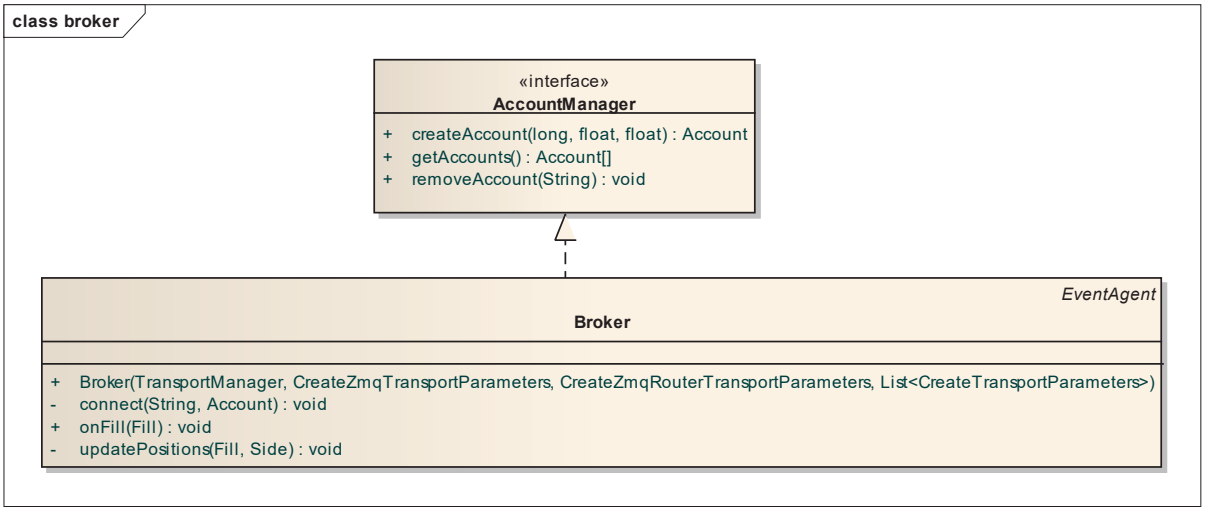

*Obrázek 10.9: Třídní diagram obecných částí modulu ATS platform*

Dále uvedu jednotlivé agenty systému a nastíním způsob jejich implementace.

### 9.6.1 Broker

Hlavním úkolem brokera je správa účtů obchodníků, konkrétně jejich strategií, které se pokaždé vážou ke svému individuálnímu účtu. Proto třída `Broker` implementuje rozhraní `AccountManager`, jak je znázorněno v diagramu na obrázku 10.10. To definuje 3 základní operace nad účty. Operaci pro založení účtu (`createAccount(long,float,float)`), která přijímá ID vytvářeného účtu, výši počátečního vkladu a výši transakčního poplatku. Uvnitř operace je volána privátní metoda `connect(String, Account)` třídy `Broker`, která slouží k propojení strategie (jednoznačně identifikované první parametrem) s nově vytvořeným účtem. Dále rozhraní definuje operaci pro vrácení všech brokerem vedených účtů (`getAccounts()`) a nakonec operaci pro odstranění existujícího účtu (`removeAccount(String)`) na základě předávaného jednoznačného názvu strategie obchodníka.

Třída `Broker` rovněž překrývá metodu `onFill(Fill)` ze třídy `EventAgent`, ve které pak volá operaci pro současnou aktualizaci stavu pozice a účtu určité strategie – `updatePositions(Fill, Side)`.




*Obrázek 10.10: Třídní diagram agenta Broker*

## 9.6.2 Trader

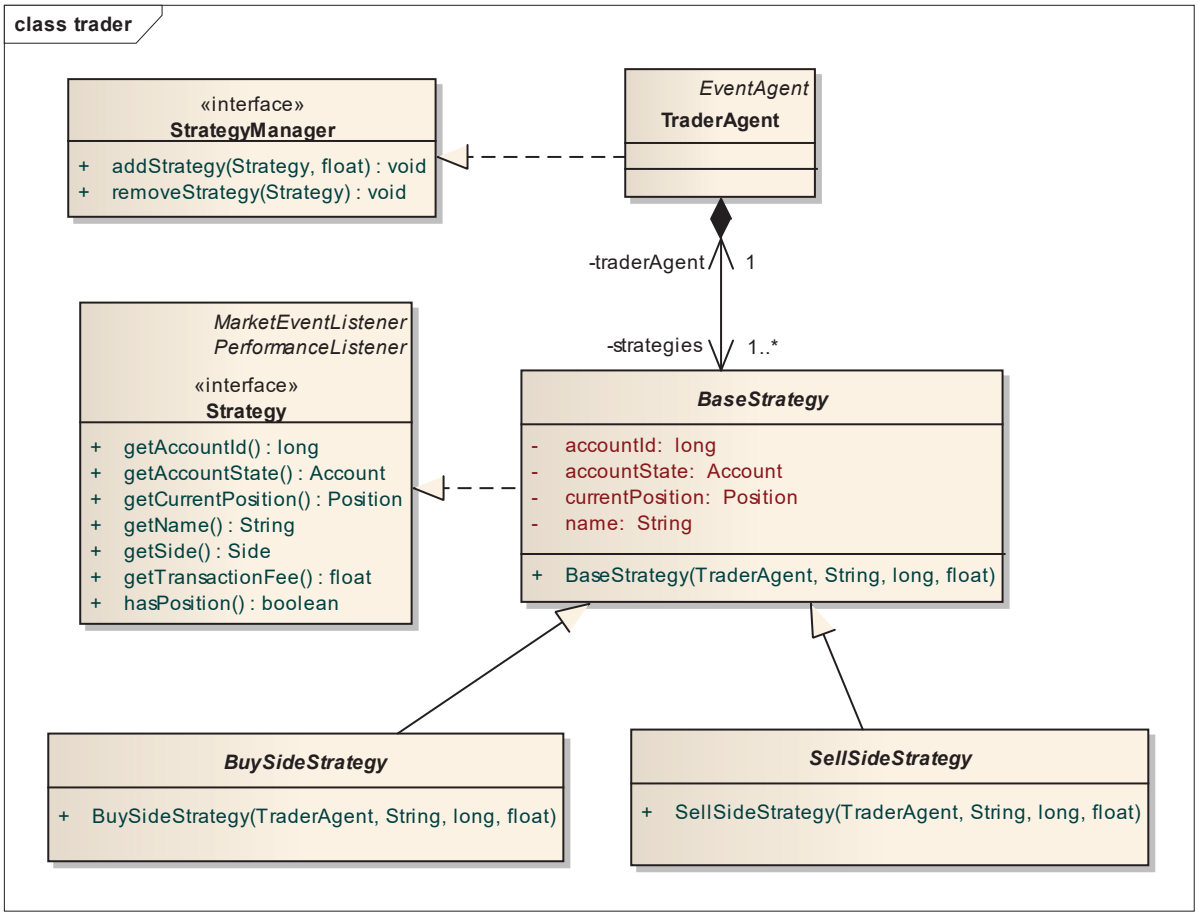
S brokerem přímo komunikuje další agent – Trader neboli obchodník. Obchodník je blíže popsán diagramem na obrázku 10.11.

Každý obchodník je reprezentován třídou `TraderAgent`, která k obchodování využívá strategie. Obchodník může současně používat více strategií. Proto implementuje rozhraní `StrategyManager` a jeho metodu pro přidávání strategií `addStrategy(Strategy, float initialBalance)`, která ihned po vyvolání posílá příkaz na registraci a svázání registrace s novým účtem brokerovi, jak bylo popsáno výše. Aby byla registrace vůbec možná, musí každá strategie obsahovat určité informace. Ty jsou definovány rozhraním `Strategy` a jeho metodami. Metoda `getAccountId()` vrací identifikátor účtu, `getName()` vrací jedinečnou identifikaci strategie a `getTransactionFee()` vrací výši poplatků za každou obchodní transakci. Ostatní metody slouží k získání aktuálního stavu strategie a pro tzv. Money-management.

Druhá metoda rozhraní `Strategy - removeStrategy(Strategy)` – slouží k odstranění již zaregistrované strategie. Obdobně jako metoda pro registraci posílá tato operace příkaz k odregistrování (událost `UnregisterStrategy`) také Brokerovi, který si ji touto cestou může rovněž odregistrovat.

Rozhraní `Strategy` dále rozšiřuje `MarketEventListener` a `PerformanceListener`, a tedy podobně jako `EventAgent` má schopnost odchytávat obchodní události pro něj určené.

Strategie mohou být obecně dvojího druhu, které jsou blíže modelovány abstraktními třídami `SellSideStrategy` a `BuySideStrategy`. Zatímco třída `SellSideStrategy` představuje strategie takzvaných tvůrců trhu (*Market Maker*), poskytovatelů likvidity na trhu, třída `BuySideStrategy` je základem pro strategie běžných obchodníků (*Market Taker*).




*Obrázek 10.11: Třídní diagram agenta Trader*

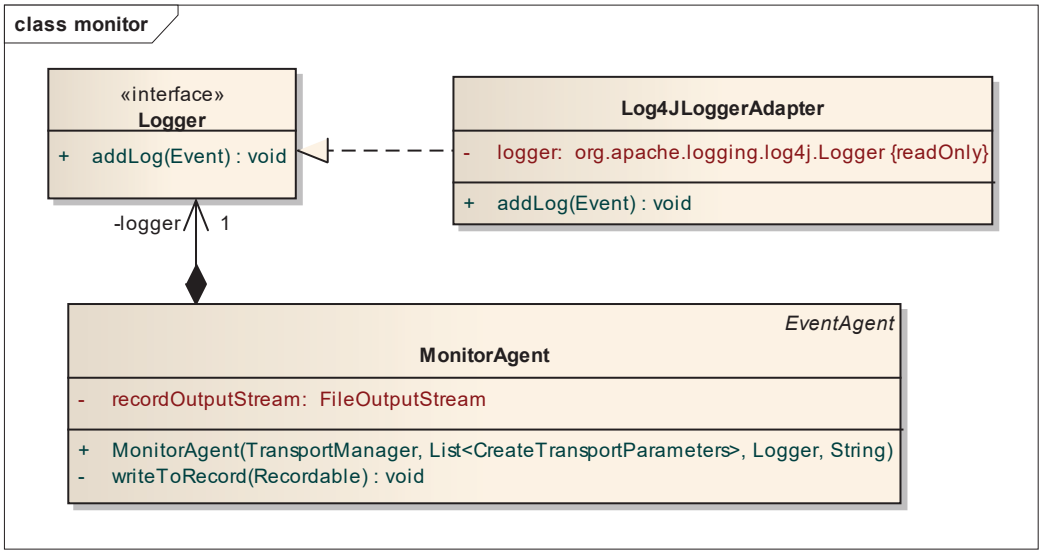
## 9.6.3 Monitor

Pro monitorování všech událostí platformy je připravena třída `MonitorAgent` znázorněna na obrázku 10.12. Stejně jako ostatní komponenty dědící ze třídy `EventAgent` může odchytávat všechny události platformy, které pak předává instanci typu `Logger`. Rozhraní `Logger` definuje jedinou metodu `addLog(Event)` pro zalogování předáváné události. Doposud existuje jediná implementace ve třídě `Log4JLoggerAdapter`, která je postavena na knihovně Log4j2. Konfigurace výstupu pro logování je pak realizována standardním konfiguračním souborem pro log4j2 – log4j2.xml. Ten musí být umístěn v classpath[9] projektu.

`MonitorAgent` se kromě logování stará rovněž o zaznamenávání historie obchodování agentů, která může později sloužit k analýze časové řady a událostí akciového trhu. Záznam je ukládán do souboru zavoláním metody `writeToRecord(Recordable)`. Název souboru pro záznam je předáván jako parametr kontstruktoru třídy. Rozhraní `Recordable` je povinně implementováno každou třídou

[9] Parametr pro nastavení umístění uživatelsky definovaných tříd a balíčků. V Maven projektu jsou do classpath umístěny také soubory ze `src/main/resources`

současně implementující rozhraní `TransportObject`, a tak je zajištěno, že každá událost platformy tuto vlastnost splňuje.

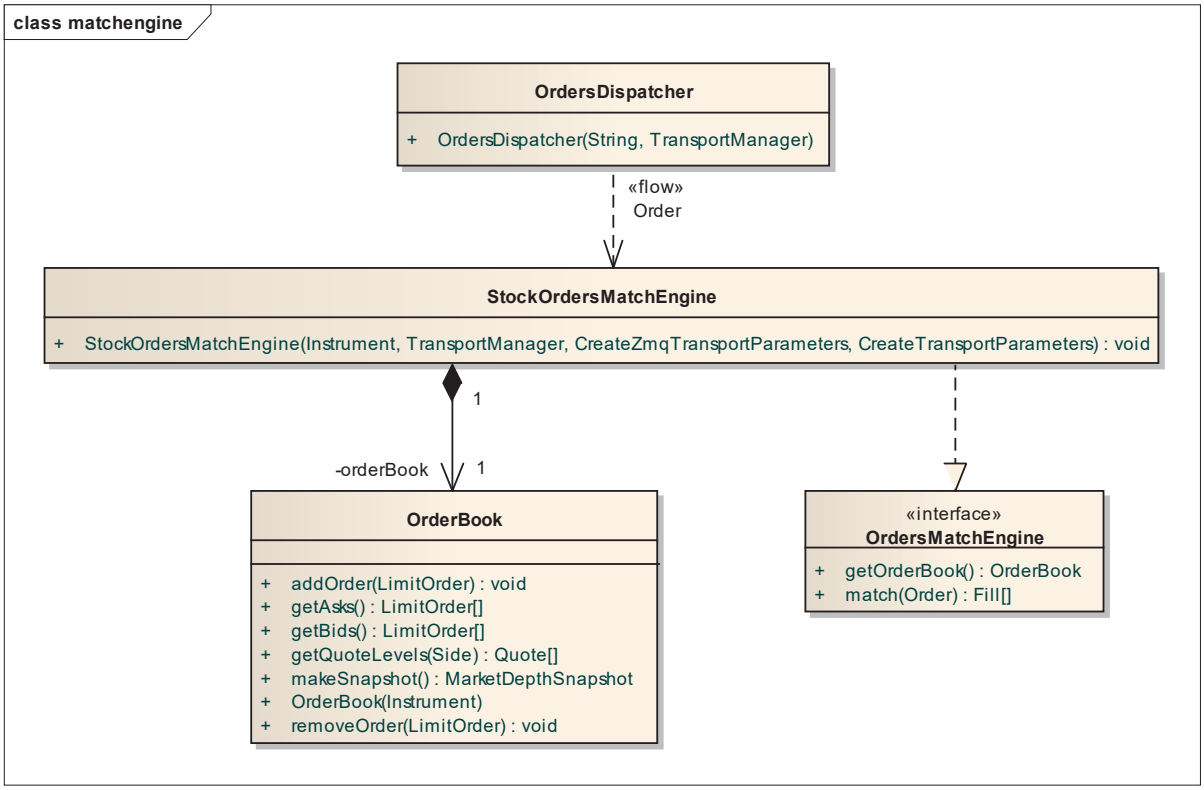



*Obrázek 10.12: Třídní diagram agenta Monitor*

### 9.6.4 Orders Dispatcher

Posledním typem agenta platformy je třída `OrdersDispatcher` (Obrázek 10.13). Ten se od předchozích typů agentů odlišuje tím, že nedědí ze třídy `EventAgent`. To je logické, jelikož většinu touto třídou odchytávaných událostí sám Orders Dispatcher, potažmo Orders Match Engine generuje. A události generované agentem typu Broker zase Orders Dispatcher nezajímají. Orders Dispatcher zde vystupuje v roli brány, která na základě podkladového aktiva (definovaného v instanci třídy `Instrument`) přeposílá příchozí příkazy `Order` dále do odpovídajících implementací rozhraní `OrdersMatchEngine`. Protože diplomová práce uvažuje pouze o akciových trzích, jedinou dosavadní implementací tohoto rozhraní je třída `StockOrdersMatchEngine`.

Každá implementace rozhraní `OrdersMatchEngine` musí povinně implementovat metodu `match(Order)` pro vypořádávání příchozích příkazů. Výstupem jsou události typu `Fill`, které popisují vypořádání vstupního příkazu. Může jich být současně více pro jeden vstupní příkaz, ale nemusí dojít také k žádnému plnění. Každý `Fill` popisuje vypořádání příchozího příkazu k jednomu jedinému příkazu doposud uchovávanému v Order Booku. Každý `OrdersMatchEngine` si tedy musí držet odkaz na instanci třídy `OrderBook`, která právě slouží k uchovávání nevypořádaných příkazů typu Limit. `OrderBook` kromě práce s příchozími příkazy (metody `addOrder(LimitOrder)`, `getAsks()`, `getBids()`, `removeOrder(LimitOrder)`) definuje metodu pro získání aktuálních cenových úrovní Order Booku (`getQuoteLevels(Side)`) a metodu pro zveřejnění svého okamžitého celkového stavu – `makeSnapshot()`.




*Obrázek 10.13: Třídní diagram agenta OrdersDispatcher*

# 10 Testování výkonnosti platformy

Protože nejdůležitějším faktorem při testování výkonnosti obchodních strategií je rychlost, s jakou mohou být tyto test provedeny a vyhodnoceny, budu se v následující kapitole věnovat testování rychlosti navržené platformy.

Budeme měřit dobu vyřízení příkazu, tedy čas, který bude trvat od odeslání požadavku obchodníkem, přes jeho přijmutí odpovídajícím Match Enginem, zpracováním, až po přijetí vypořádání zpět obchodníkem. Tomuto koloběhu budeme říkat *roundtrip*. Započítávat budeme pouze příkazy typu Market, u nichž dochází k okamžité exekuci. Může dojít k jeho částečnému nebo úplnému vypořádání (s ohledem na použité strategie dojde k některému z plnění téměř pokaždé). V případě příkazu typu Limit dochází v našem testovaném příkladu nejprve k zařazení do Order Booku a jeho započítávání by mohlo zkreslit výsledky. K zapisování času, kdy dojde k zachycení události vypořádání obchodníkem, využijeme jednu z listener metod `onFill(Fill)` zmíněnou v předchozí kapitole.

Testování opřeme o jednoduché strategie, které co možná nejrychleji reagují na nastalé změny trhu a které neustále posílají další příkazy k vypořádání. Jejich logika je implementována ve třídách `PrimitiveMarketMaker` a `BrutalMarketTaker`.

Začnu nejprve s vysvětlováním logiky ve třídě `BrutalMarketTaker`, která po spuštění jednoduše chrlí jeden příkaz za druhým. Vše jsou příkazy typu Market, aby došlo pokud možno k plnění příkazu. Sice nedochází k zaplňování Order Booku ze strany tohoto obchodníka, ale o to nám zde nejde. Netestujeme rychlost vyhledávání a sortování v `TreeSetu`[10], s jehož pomocí jsem Order Book naimplementoval a u kterého víme, že složitost operací je logaritmická, nýbrž se zaměřujeme na rychlost celé platformy, potažmo na rychlost transportní vrstvy. Ta zde hraje hlavní roli a má hlavní podíl na latenci.

Protože bez likvidity by k žádnému obchodu nemohlo na trhu dojít, stojí na straně poskytovatele likvidity třída `PrimitiveMarketMaker`. Její logika je také vcelku jednoduchá. Na předem nastavených cenových úrovních si na obou stranách trhu vztyčí vždy po jednom příkazu typu Limit. Při jakémkoli plnění je pak bezodkladně doplňuje stejným množstvím podkladového aktiva, jako bylo vypořádáno.

Každý test bude spuštěn po dobu 60 sekund, během kterých se na začátku nejprve každý z obchodníků zaregistruje k brokerovi pro vedení účtu, a poté nezávisle na druhém obchodníkovi začne posílat příkazy do trhu. Příkazy obchodník začne posílat dokonce nezávisle na výsledku registrace u brokera. Skutečné odeslání takových příkazů do trhu je však opožděno samotnou implementací třídy `TraderAgent`, která je odešle až ve chvíli, kdy je dokončena a úspěšně potvrzena registrace účtu.

Testování proběhlo na několika různých konfiguracích výpočetních zařízení a prostředí:

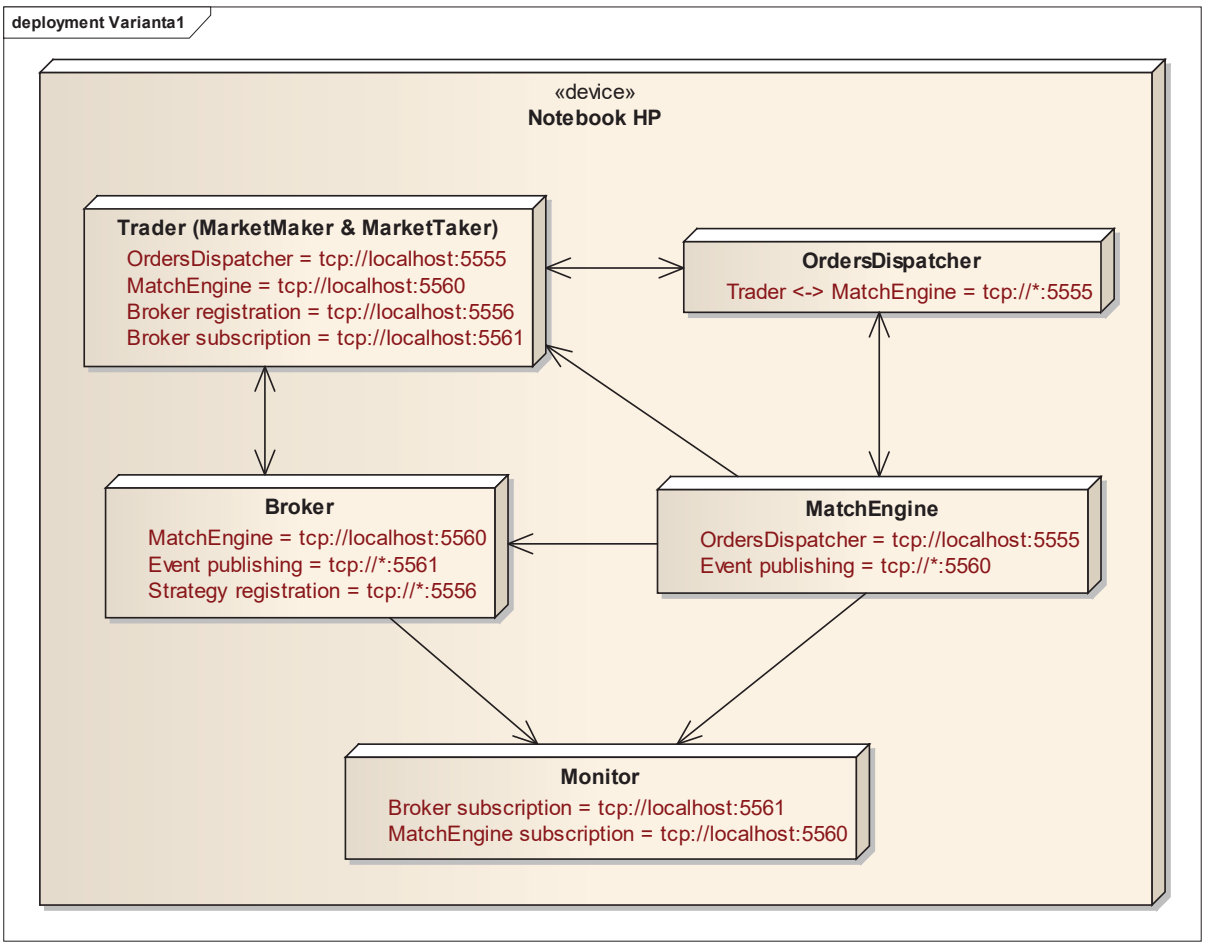
- notebook HP s 4-jádrovým procesorem Intel Core i7-4700MQ @ 2,40GHz (8 logických procesorů) a 32GB RAM, operační systém Windows 10. Aplikace však spuštěna na virtualizovaném operačním systému Ubuntu 14.04 LTS s přidělenými 16GB operační paměti, Java 1.8.0_25,

[10] z balíčku `java.util` standardní edice Javy

- netbook Asus Eee s 2-jádrovým procesorem Intel Atom N450 @ 1,66GHz, 2GB RAM, Ubuntu 14.04 LTS, Java 1.8.0_77,
- 84-jádrový počítač s Intel Xeon CPU E5-4610 0 @ 2,40GHz, 1TB RAM, SUSE Linux Enterprise Server 12, Java 1.8.0_73.

## 10.1 Varianta 1

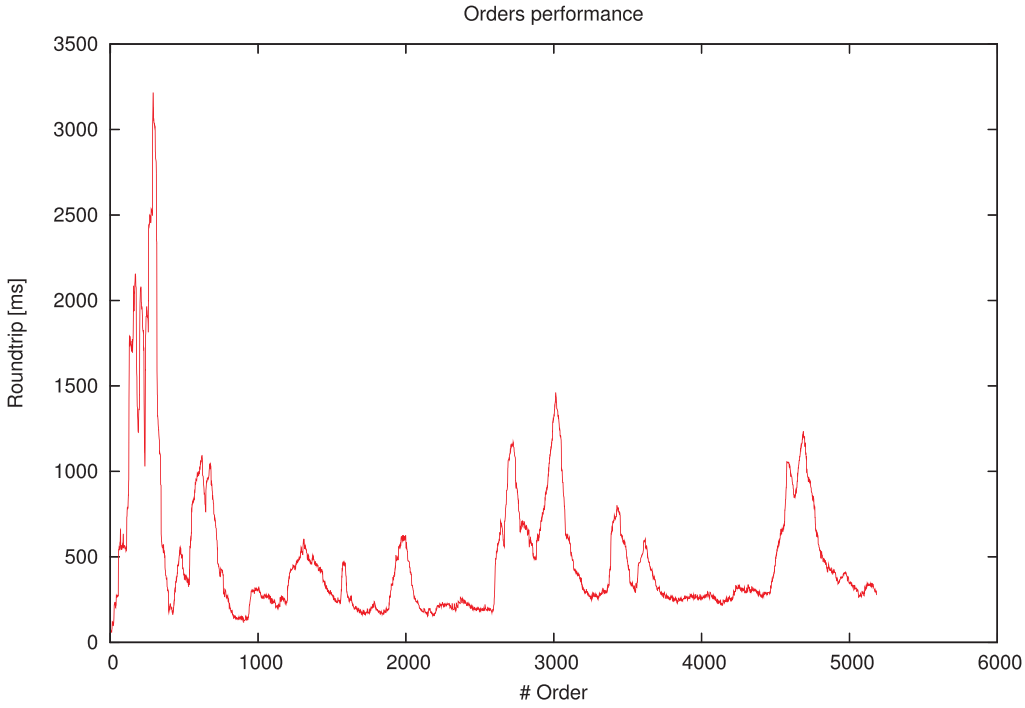
První varianta plně v režii prvně zmíněného 4-jádrového notebooku od HP. Nastavení komunikačních rozhraní jednotlivých agentů blíže znázorňuje následující schéma (Obrázek 11.1).




*Obrázek 11.1: Schéma zapojení varianty 1 pro testování výkonnosti*

Šipky ve schématu znázorňují komunikační cesty a přenos dat od jednoho agenta systému k druhému. Jedna šipka mezi týmiž agenty může znázorňovat současně více komunikačních kanálů, jedna šipka je použita pouze z důvodu přehlednosti. Tak je tomu například v případě komunikace mezi agenty Trader a Broker, kdy Trader jednak posílá Brokerovi žádosti o registraci strategie a přijímá potvrzení těchto

žádostí jednou komunikační cestou, ale současně se Trader registruje pro odběr obchodních dat poskytovaných Brokerem jinou komunikační cestou.
Jedno spojení bod-bod je definováno vždy dvojicí se stejným číslem portu - např. `tcp://*:5555` pro stranu, která naslouchá na daném portu, a `tcp://localhost:5555`, která se na tuto službu připojuje.
Oba agenti vystupující zde jako obchodníci – Market Maker i Market Taker – mají totožnou konfiguraci připojení, a proto je pro přehlednost uvedena také jen jednou.
Výsledky varianty 1 jsou uvedeny na obrázku 11.2.

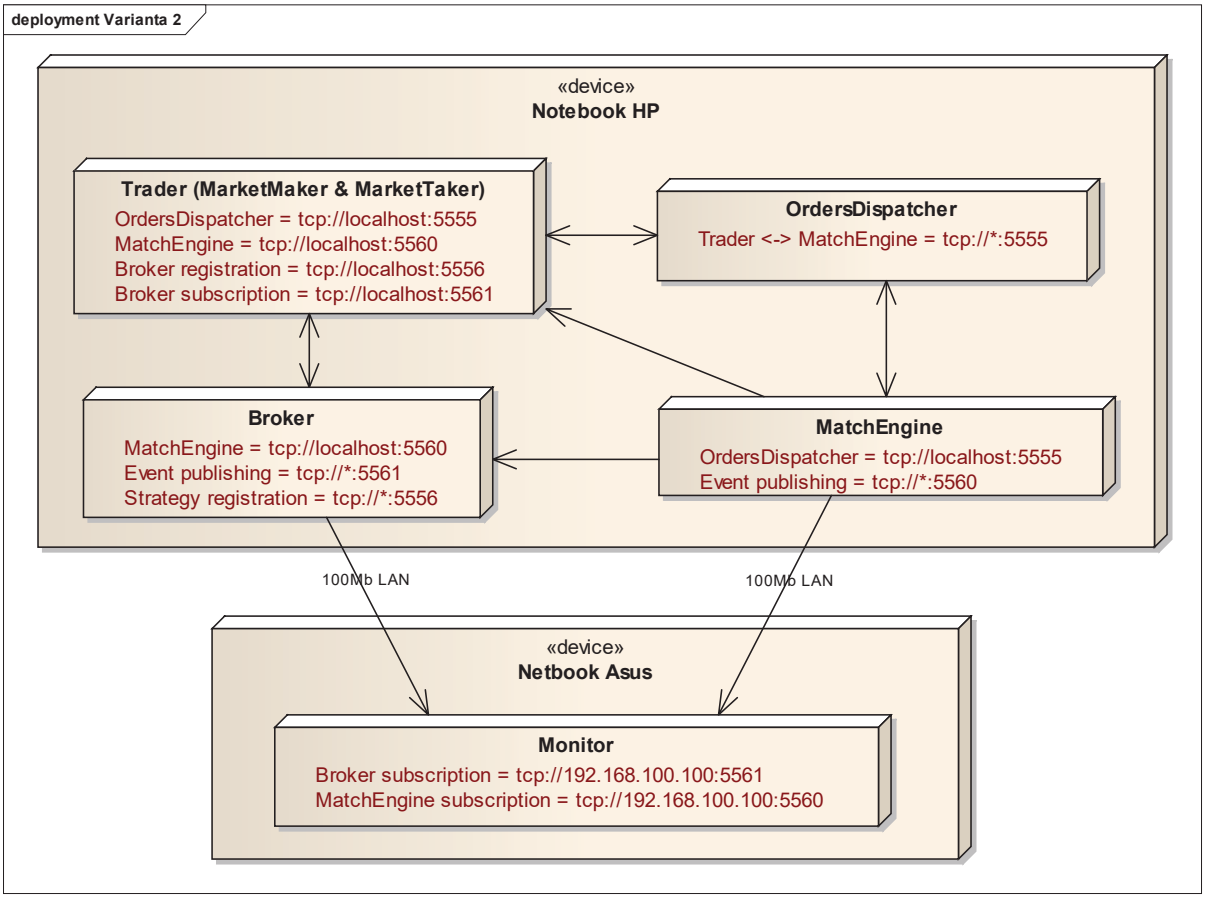



*Obrázek 11.2: Graf doby trvání roundtripu příkazu pro variantu 1*

Průměrná rychlost zpracování jednoho příkazu činí přibližně **489 ms**.
Z výsledků je patrné, že dochází k celkem velkým výkyvům rychlosti zpracování, hlavně na začátku grafu. Zřejmě se zde projevuje nedostatek výkonu, který je u varianty 1 poskytován. Každé socketové připojení totiž vyžaduje 1 vlákno, 1 vlákno vyžaduje pro svůj chod rovněž každý agent samotný, několik vláken je také využíváno pro výpočetní část (Match Engine, Broker) a něco si vezme režie kolem. Ve výsledku je to něco přes 30 vláken, které by měly běžet téměř neustále paralelně. Pravděpodobně se tak projevuje horší management vláken v Javě, který nestíhá dostatečně rychle a vhodně přepínat mezi jednotlivými vlákny vyžadujícími v daný moment procesorový čas.
Ve variantě 2 se pokusíme využít další počítač k lepší distribuci práce využitím více výpočetních jader.

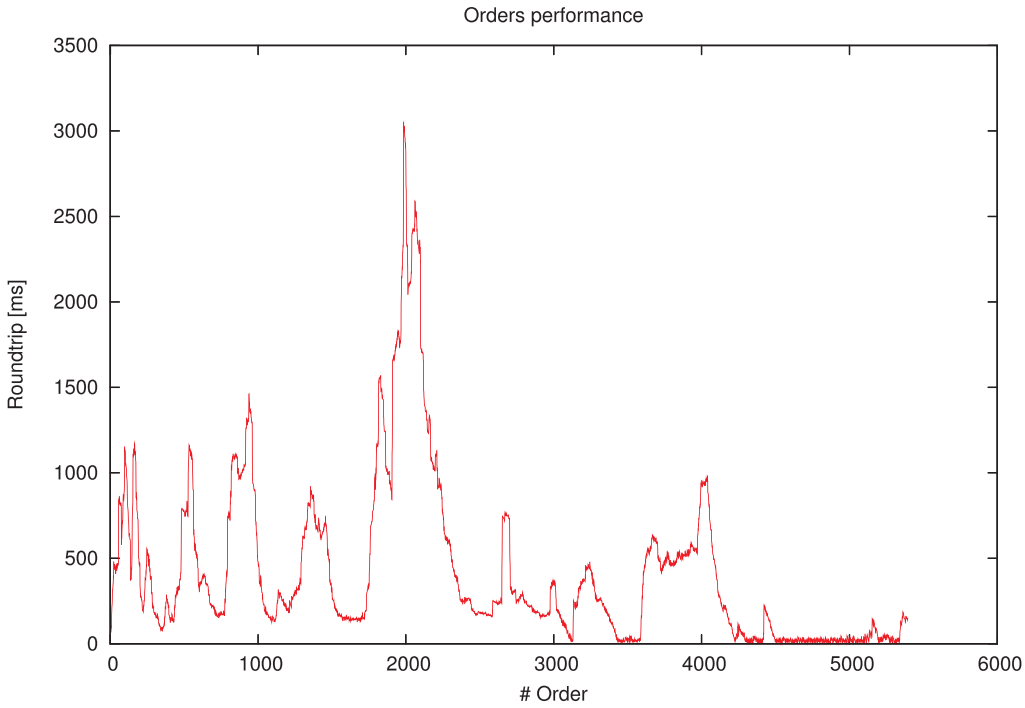
## 10.2 Varianta 2

Ve variantě 2 přidáme další výpočetní stroj, na který přeneseme část práce a odlehčíme tak prvnímu počítači. Protože však Netbook Asus nenabízí bůhví jak velký výpočetní výkon, využijeme jej pouze pro monitoring. Schéma zapojení je k dispozici na obrázku 11.3 dále.




*Obrázek 11.3: Schéma zapojení varianty 2 pro testování výkonnosti*

Pro přenesení agenta pro monitoring na druhý počítač bylo nutno pouze změnit nastavení dvou socketových připojení pro odběr událostí platformy proudící z komponent Broker a Match Engine. Monitor agent se tak připojuje na vzdálené rozhraní notebooku HP identifikované IP adresou 192.168.100.100. Mezi oběma počítači je vytvořena 100Mb lokální síť pomocí kabelového připojení, aby docházelo co možná k nejmenším latencím připojení. Netbook Asus na druhé straně tohoto spojení má IP adresu 192.168.100.101, kterou však není nutno nikde uvádět. Netbook neposkytuje žádnou službu ostatním, ke které by se mohli připojovat.
Výsledky měření pro variantu 2 jsou k dispozici na následujícím obrázku 11.4.

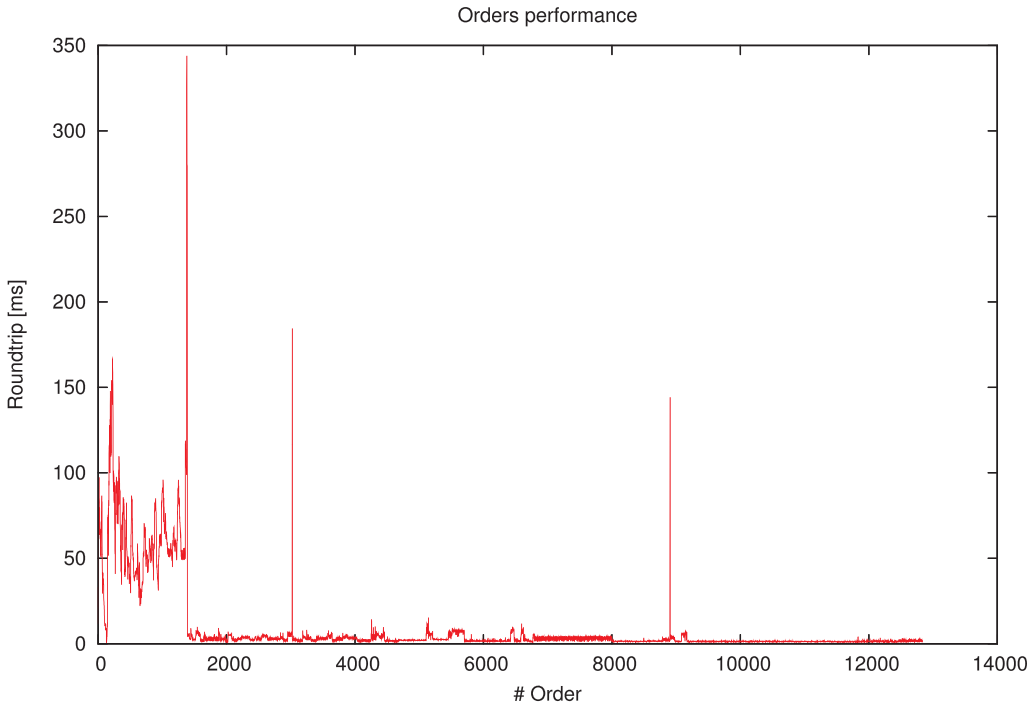



*Obrázek 11.4: Graf doby trvání roundtripu příkazu pro variantu 2*

Přínos distribuce výpočtů mezi více počítačů je zde patrný. Průměrná rychlost zpracování jednoho příkazu činí nyní přibližně **421 ms**. Ke konci spuštěného testu jsou odezvy téměř lineární a vcelku s nízkými hodnotami, přesto jsou stále patrné četné nepravidelnosti v podobě ostrých peaků. Poměr souběžně běžících výpočetních vláken ku potřebným je stále v náš neprospěch přibližně 10:30.
Ve variantě 3 se pokusíme tento nepříznivý stav zvrátit opravdu velkou výpočetní silou v podobě 84 současně dostupných vláken, které by měly zcela pokrýt naši potřebu.

## 10.3 Varianta 3

Varianta 3 nabízí nekompromisní výkon v podobě 84 současně běžících jader. Schéma zapojení je stejné jako v případě varianty 1 (Obrázek 11.1), a tak jej zde nebudu znovu uvádět.
Výsledky měření jsou uvedeny na obrázku 11.5 dále.




*Obrázek 11.5: Graf doby trvání roundtripu příkazu pro variantu 3*

Až na pár výjimek je průběh grafu téměř lineární a pohybuje se pod 9 ms na jedno zpracování. Celých 11437 z 12837 příkazů je vyřízeno rychleji než za 9 ms. Průměrná doba je zde něco málo přes **9 ms**. Průměrná doba je však značně ovlivněna prvními příkazy, u nichž doba vyřízení činí i přes 300 ms. Jedná se pravděpodobně o příkazy, jejichž odeslání do trhu je opožděno do doby, než je potvrzena registrace obchodníka u Brokera, avšak jsou již zadány obchodníkem a tedy jejich časové razítko je u nich již rovněž vygenerováno. Tento fakt se ostatně projevuje ve větším či menším měřítku u všech měřených konfigurací prostředí. Také celkový náběh systému zde může hrát svoji roli.
Neméně důležitým faktem je, že běžně dostupné JVM[11] nejsou optimalizovány pro běh low-latency aplikací. [23] Některé finanční instituce dokonce využívají speciálně upravených, komerčně dostupných JVM. [24]

[11] Java Virtual Machine – softwarový stroj, program, který se stará o běh aplikací v Javě. Interpretuje pro operační systém instrukce v tzv. Java byte kódu.

# 11 Závěr

Cílem práce bylo navrhnout a vytvořit platformu, v níž bude možné simulovat běh akciového trhu, což bylo splněno. Předem jsem se musel podrobněji seznámit s fungováním trhu, s jeho základními principy a částmi. Do detailu jsem musel prozkoumat, jakými přístupy dochází k vykonávání a hlavně vypořádání obchodních příkazů. Při pohledu zpět mi přijde celý mechanismus logický a vychází z potřeb trhu – především pak jednotlivé algoritmy pro vypořádání příkazů. Pro implementaci platformy jsme zvolili základní typ algoritmu pro akciové trhy FIFO, založený na čistě časové prioritě a způsobu plnění. Platformu je díky definovaným rozhraním jednoduché rozšířit o další algoritmy pro zpracování obchodních příkazů, toto je však ponecháno jako námět k dalšímu vývoji práce.

Vytvořili jsme platformu, která nabízí obchodování akcií v reálném čase a to na základě naimplementovaného chování obchodníků, konkrétně jejich obchodních strategií. Implementace souboru komplexních strategií by mohla být rovněž námětem pro další vývoj tématu práce. Konkrétním typem implementované strategie by mohlo být párové obchodování uvedené v kapitole 3.4.1.

Byl vytvořen Event-driven systém založený na asynchronním modelu zpracování událostí platformy, který nabízí responzivní neblokující chování systému. Události trhu, potažmo celé platformy, definují ontologii[12] účastníků trhu – obchodník, broker, dispečer obchodních příkazů, Match Engine. Ontologie je fundamentem multiagentních systému. Toto vcelku moderní paradigma jsme využili při návrhu platformy. Navržený systém tak může mít rovněž síťový charakter, v němž dochází k interakci jednotlivých uzlů v síti. Takový přístup umožňuje vcelku neomezené možnosti škálování celého řešení prostřednictvím distribuce výpočtů na více počítačů v síti. Každý agent a jeho komplexní výpočetní logika může být přenesen na libovolný počítač v síti pro využití vyššího výkonu při simulaci. Odtud se pak prostřednictvím síťových rozhraní (socketů) připojuje k ostatním agentům, s nimiž potřebuje komunikovat. To se nám ostatně osvědčilo i při testování výkonnosti.

Dalším splněným bodem zadání je monitorování událostí systému a jejich ukládání. Ukládáním událostí jsme dosáhli další užitečné vlastnosti – Event sourcing. Ten může sloužit nejen pro analýzu časové řady trhu, ale také jako žurnál pro zpětné odhalování neočekávaných událostí. Postupným aplikováním příkazů od začátku simulace, anebo z důvodu rychlosti od posledního Snapshotu trhu, jsme schopni dosáhnout aplikačního stavu před takovou událostí a tak ji přesně replikovat. Jsme schopni rozlišit stav aplikace v jakémkoli bodu v čase.

Posledním bodem práce bylo otestování rychlosti platformy. Pro testování byly naimplementovány dvě jednoduché strategie – jedna pro Market Makera, druhá pro Market Takera. V prvních dvou variantách konfigurace testovacího prostředí se negativně projevil nedostatek výpočetního výkonu, konkrétně nedostatek procesorových jader. V poslední třetí měřené variantě jsme pravděpodobně narazili na výkonnostní strop Javy a jejího JVM. Slibně se nám jeví Disruptor pattern od společnosti LMAX, který by svým řešením mezi-vláknové komunikace mohl zmíněné problémy zmírnit, možná snad úplně vyřešit.

[12] komunikační jazyk

# Literatura


[1] Ondřej Hartman, tým FXstreet.cz. *Začínáme na burze*. BizBooks, 2013. ISBN 978-80-265-0033-9.

[2] Martin Sewell. *Characterization of Financial Time Series*, UCL Research Note RN/11/01, 2011.

[3] Marc Potters, Jean-Philippe Bouchaud. *More statistical Properties of Order Books and Price Impact*, Physica A, 2003.

[4] Supakorn Mudchanatongsuk, James A. Primbs and Wilfred Wong. *Optimal Pairs Trading: A Stochastic Control Approach*, 2008.

[5] Karl Sigman. *Geometric Brownian motion*. Columbia University, 2006

[6] *Burza cenných papírů Praha*. http://www.pse.cz/dokument.aspx?k=Profil-Burzy.

[7] G. E. Uhlenbeck and L. S. Ornstein. *On the theory of Brownian motion*. University of Michigan, 1930.

[8] M. C. Wang and G. E. Uhlenbeck. *On the theory of Brownian motion II*. University of Michigan, 1945.

[9] J. L. Doob. *The Brownian Movement and Stochastic Equations*. 1942.

[10] L. Breiman. *Probability*. Addison-Wesley, 1968. ISBN 978-0-898-71296-4.

[11] I. Karatzas and S. E. Shreve. *Brownian Motion and Stochastic Calculus*. Springer-Verlag, 1988. ISBN 978-0-387-97655-6.

[12] Mahdavi Damghani B. *The Non-Misleading Value of Inferred Correlation: An Introduction to the Cointelation Model*, 2013.

[13] Fabio Luigi Bellifemine, Giovanni Caire, Dominic Greenwood. *Developing Multi-Agent Systems with JADE*. Wiley, 2007. ISBN 978-0-470-05747-6.

[14] Rajeev Ranjan. *Order Matching Algos*. [online] . <https://sites.google.com/site/rajeevranjansingh/order-matching-execution-side>.

[15] Pieter Hintjens. *ZeroMQ: Messaging for Many Applications*. O'Reilly Media, Inc., 2013.

[16] Vladimir I. Arnol'd. *Catastrophe Theory*. 1984. ISBN 978-3-642-96799-3.

[17] *HTTP/2 FAQ*. [online]. <https://http2.github.io/faq/#why-is-http2-binary>.

[18] *Protocol Buffers*. [online]. <https://developers.google.com/protocol-buffers/docs/overview>.

[19] *JVM Serializers Benchmarking.* [online]. <https://github.com/eishay/jvm-serializers/wiki>.

[20] *Log4j2*. [online]. <http://logging.apache.org/log4j/2.x/>.

[21] *Log4j2 – Comparison*. [online]. <http://logging.apache.org/log4j/2.x/manual/async.html#Asynchronous_Throughput_Comparison_with_Other_Logging_Packages>.

[22] *LMAX Disruptor*. [online].
<http://lmax-exchange.github.io/disruptor/>

[23] *Oracle Java virtual machine putting the brakes on high frequency trading, says Azul CEO*. [online].
<http://www.computerworlduk.com/news/it-vendors/oracle-java-virtual-machine-putting-brakes-on-high-frequency-trading-says-azul-ceo-3452852/>

[24] Azul Systems. Zing JVM. [online].
<https://www.azul.com/solutions/capital-market/>

# Příloha na CD

- Zdrojové soubory simulační platformy
- Programátorská/uživatelská dokumentace
- Data a nástroje k testovacím případům